ISSN 0265-2919

80p 32

THE HOME COMPUTER ADVANCED COURSE

MAKING THE MOST OF YOUR MICRO

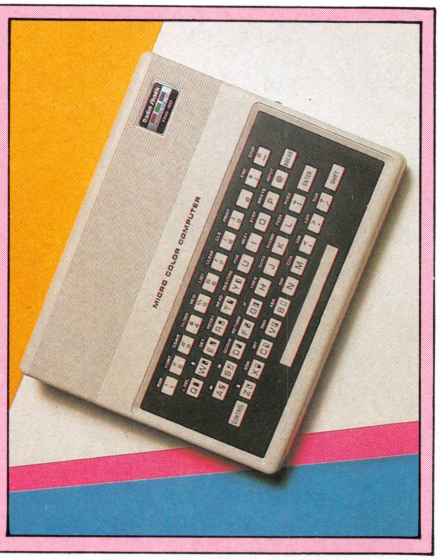
An ORBIS Publication

IR £1 Aus $1.95 NZ $1.25 SA R1.95 Sing $4.50 USA & Can $1.95

CONTENTS

Next Week

- **Continuing our occasional series on portable computers, we compare lap-helds from Olivetti, NEC and Tandy — different versions of the same basic computer.**
- **Our robotics series investigates and simulates the robot arm.**
- **We introduce you to the dynaturtle, in a LOGO-based guided tour of vector algebra.**



QUIZ

1) What does 'hard-sectored' mean?
2) Where would you expect to find a MIRROR command?
3) What is 'commonality'?
4) Where do you find 'metal collar workers'?

Answers To Last Week's Quiz

1) 'Elektro' was an early robot, built by Westinghouse. It was seven feet tall, and could count, walk, talk and distinguish colours.

2) 'Stubs' are sections of an uncompleted program that simulate the action of as yet unwritten routines, permitting testing of other parts of the program.

3) A 'high-pass filter' is an electronic filter. It passes only those signals whose frequency is higher than a pre-set lower limit.

4) 'CMOS' stands for 'complementary metal oxide semi-conductor'. These 'field-effect transistors' take less power than conventional chips, so are often used in battery-powered computers.

Editor Mike Wesley; **Art Director** David Whelan; **Technical Editor** Brian Morris; **Production Editor** Catherine Cardwell; **Art Editor** Claudia Zeff; **Chief Sub Editor** Robert Pickering; **Designer** Julian Dorr; **Art Assistant** Liz Dixon; **Editorial Assistant** Stephen Malone; **Sub Editor** Steve Mann; **Researcher** Melanie Davis; **Staff Writer** Steve Colwill; **Contributors** Geoff Bains, Harvey Mellor, Mike Curtis, Steve Colwill, Chris Naylor, Max Phillips, Matt Nicolson, Steve Malone; **Software Consultants** Pilot Software City; **Group Art Director** Perry Neville; **Managing Director** Stephen England; **Published by Orbis Publishing Ltd: Editorial Director** Brian Innes; **Project Development** Peter Brookesmith; **Executive Editors** Chris Cooper, Maurice Geller; **Production Controller** Peter Taylor-Medhurst; **Circulation Director** David Breed; **Marketing Director** Michael Joyce; **Designed and produced by Bunch Partworks Ltd; Editorial Office** 14 Rathbone Place, London W1P 1DE; © **APSIF Copenhagen 1984; © Orbis Publishing Ltd 1984: Typeset by Universe; Reproduction by Mullis Morgan Ltd; Printed in Great Britain by Artisan Press Ltd, Leicester**

HOME COMPUTER ADVANCED COURSE – Price UK 80p IR £1.00 AUS $1.95 NZ $2.25 SA R1.95 SINGAPORE $4.50 USA and CANADA $1.95

How to obtain your copies of HOME COMPUTER ADVANCED COURSE – Copies are obtainable by placing a regular order at your newsagent, or by taking out a subscription. Subscription rates: for six months (26 issues) £23.80; for one year (52 issues) £47.60. Send your order and remittance to Punch Subscription Services, Watling Street, Bletchley, Milton Keynes, Bucks MK2 2BW, being sure to state the number of the first issue required.

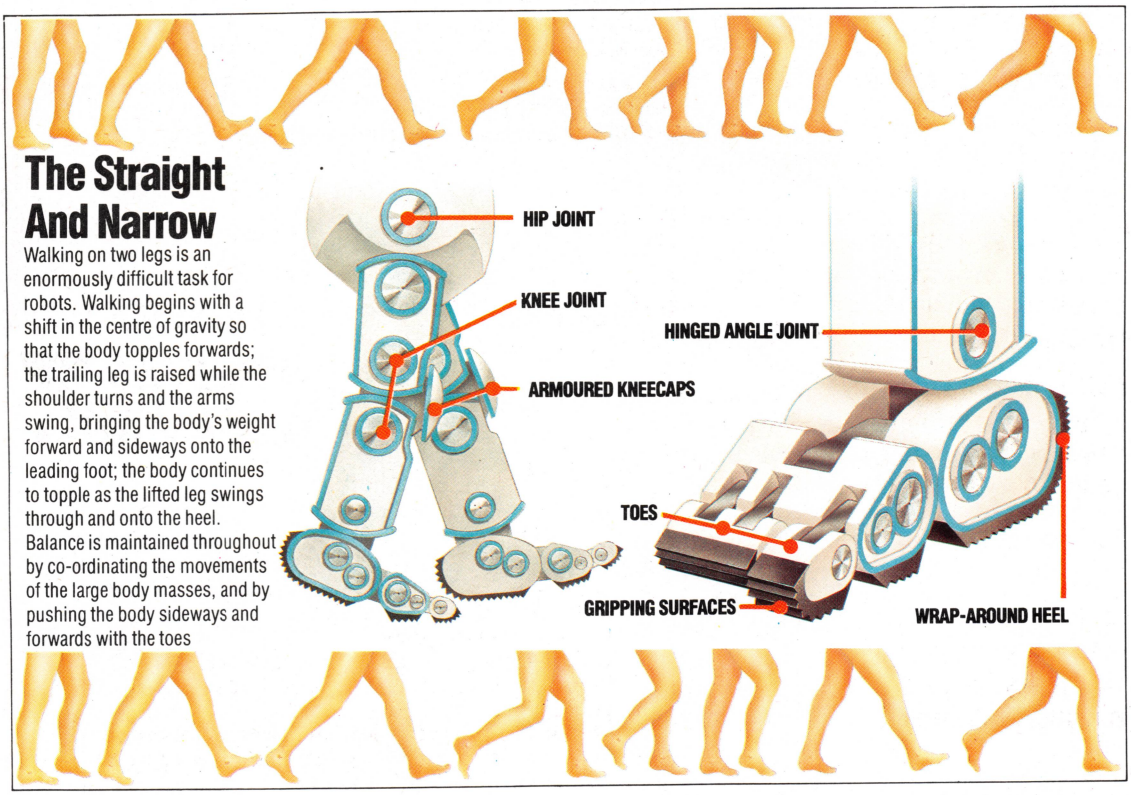
Back Numbers UK and Eire – Back numbers are obtainable from your newsagent or from HOME COMPUTER ADVANCED COURSE. Back numbers, Orbis Publishing Limited, 20/22 Bedfordbury, LONDON WC2N 4BT at cover price. AUSTRALIA: Back numbers are obtainable from HOME COMPUTER ADVANCED COURSE. Back numbers, Gordon & Gotch (Aus) Ltd, 114 William Street, PO Box 767G Melbourne, Vic 3001. SOUTH AFRICA, NEW ZEALAND, EUROPE & MALTA: Back numbers are available at cover price from your newsagent. In case of difficulty write to the address in your country given for binders. South African readers should add sales tax.

How to obtain binders for HOME COMPUTER ADVANCED COURSE – UK and Eire: Please send £3.95 per binder if you do not wish to take advantage of our special offer detailed in Issues 5, 6 and 7. EUROPE: Write with remittance of £5.00 per binder (incl. p&p) payable to Orbis Publishing Limited, 20/22 Bedfordbury, LONDON WC2N 4BT. MALTA: Binders are obtainable through your local newsagent price £3.95. In case of difficulty write to HOME COMPUTER ADVANCED COURSE BINDERS, Miller (Malta) Ltd, M.A. Vassalli Street, Valletta, Malta. AUSTRALIA: For details of how to obtain your binders see inserts in early issues or write to HOME COMPUTER ADVANCED COURSE BINDERS, First Post Pty Ltd, 23 Chandos Street, St. Leonards, NSW 2065. The binders supplied are those illustrated in the magazine. NEW ZEALAND: Binders are available through your local newsagent or from HOME COMPUTER ADVANCED COURSE BINDERS, Gordon & Gotch (NZ) Ltd, PO Box 1595, Wellington. SOUTH AFRICA: Binders are available through any branch of Central Newsagency. In case of difficulty write to HOME COMPUTER ADVANCED COURSE BINDERS, Intermag, PO Box 57394, Springfield 2137.

Note – Binders and back numbers are obtainable subject to availability of stocks. Whilst every attempt is made to keep the price of the issues and binders constant, the publishers reserve the right to increase the stated prices at any time when circumstances dictate. Binders depicted in this publication are those produced for the UK market only and may not necessarily be identical to binders produced for sale outside the UK. Binders and issues may be subject to import duty and/or local taxes, which are not included in the above prices unless stated.

COVER PHOTOGRAPHY BY MARCUS WILSON-SMITH

STEPPING OUT



The Straight And Narrow

Walking on two legs is an enormously difficult task for robots. Walking begins with a shift in the centre of gravity so that the body topples forwards; the trailing leg is raised while the shoulder turns and the arms swing, bringing the body's weight forward and sideways onto the leading foot; the body continues to topple as the lifted leg swings through and onto the heel. Balance is maintained throughout by co-ordinating the movements of the large body masses, and by pushing the body sideways and forwards with the toes

STEVE CROSS

In the first part of our series on robotics we looked at how robots have developed from science fiction fantasy to the 'metal collar' workers of today's production lines. Now we discuss the three principal methods of robot movement and the most efficient means of controlling it.

Long before a child learns to walk it will be capable of picking objects up and will demonstrate intelligence in numerous other ways, but walking is a skill that takes a long time to develop and which involves considerable practice before it becomes automatic.

Robots can be made to 'walk', but the techniques involved are very different from the methods used by people. The robot may have legs, which it can swing backwards and forwards in an approximation of a human walk, but each of these legs is equipped with a foot that has wheels on its base. These wheels are fitted with ratchets to inhibit backward movement. So a robot of this type follows a set sequence of actions as it 'walks'. The disadvantage of this method it that it is difficult to develop a way of steering the robot; it will tend to move in a forward direction only and its movements will be imprecise.

A much better solution would be to make robots walk by lifting first one leg and then the other, as humans do, instead of simply swinging each limb through a limited arc. The major problem with this approach is that the robot must be able to balance on one leg as it walks. Various solutions have been tried: these include tilting the robot's body sideways and even moving the entire torso sideways on a rail so that the robot's centre of gravity is kept directly above the leg that is bearing its weight. If such a system were developed, robots could walk in an efficient manner. In theory, a walking robot could be designed to climb stairs and bring its owner a morning cup of tea. But in practice, although a robot that can climb stairs is quite feasible, a robot that 'knew' when it had reached the top of the stairs would be harder to develop because of the extra apparatus needed to detect the top step.

An alternative approach has been to mount robots on tracks. The advantage of this system is that it allows the robot to travel over rough ground. The British Army uses tracked robots for carrying out hazardous bomb-disposal duties; these machines can manoeuvre through debris and can cover reasonably rough ground.

Tracks are robust and are easily driven, but they have two main drawbacks. The first is that, as most robots are fairly small, so the size of the tracks is small and hence large obstacles cannot be

negotiated. A full-size battle tank can climb over almost any obstacle with ease — but this is only because tanks are so large, heavy and powerful. If a tank tries to climb over an object so large that the tank's centre of gravity is moved outside the area of its tracks it will fall over. This does in fact happen occasionally when the ground is very rough. The same thing will happen to your tracked robot if it tries to climb over surfaces that are too steep.

The second disadvantage is that tracks cannot be controlled precisely. Steering is carried out by halting one track so that the other track continues running; the robot (or tank) thus moves through an arc. When this happens, the stationary track may easily slip slightly and the final position may not be what was expected. A battle tank that is driven by a person can readily correct any error of this kind but, for a robot, the necessary course corrections are considerably more complicated.

For robot control, it is obviously desirable to have a set of instructions that will always cause the robot to move to exactly the right place, facing in an exactly predictable direction. For these reasons, the most common form of robot movement uses wheels. Wheels have several obvious advantages, being simple, efficient and capable of producing a much smoother movement than legs could ever manage.

Once it has been accepted that the robot should use wheels, the only problem is exact control of the movement. Consider, for instance, a clockwork toy motor car. This runs on wheels but it is not a robot, as it has no means of 'knowing' its position at any given time. What is needed is a co-ordinate system that can be used to determine an object's position on a surface — the most common system for this purpose uses Cartesian co-ordinates. With this system it is possible to locate a robot's precise position and to specify the movements needed to move it to another defined location. All that is then needed is a device to ensure that the robot can move precisely within this frame of reference.

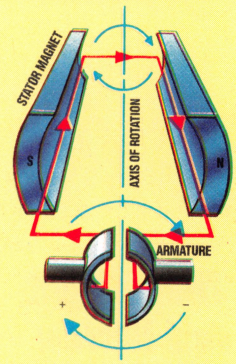
Although hydraulic or pneumatic power is occasionally used, the most common method of moving robots is via an electric motor. As we have seen in our Workshop series (pages 585 and 612), a simple electric motor can provide movement and a modest amount of control over direction. This is not suitable for precise control — a simple electric motor always turns through at least 180 degrees before coming to rest, and inertia will often cause it to rotate a little more than that.

So, for robot control, the stepper motor is normally used. This is a motor that contains a large number of coils and, although designs may vary widely, the general principle of the stepper motor allows very small, exact amounts of rotation to be specified, with little overshoot (rotating more than it should) or undershoot (rotating too little).

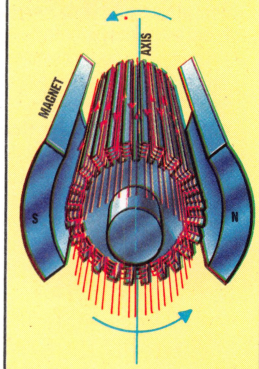
Robots that use stepper motors are widely available. Such robots often have a pen attached, allowing them to draw a line on the surface they are travelling over. These pen-wielding robots are called 'turtles', and the designs they produce are known as 'turtle graphics'. These are capable of precise movement — their accuracy may be judged by instructing them to draw a closed shape, such as a rectangle or a star, and checking to see whether or not the line drawn meets itself at the starting point.

Stepper motors and Cartesian co-ordinates can therefore give us a relatively precise method of controlling a robot's movement. However, if the robot is to do more than simply roll around in a given area, bumping into obstacles, it will need to be able to respond quickly and accurately to external conditions. We will consider this further in the next instalment.

Stepping A Measure



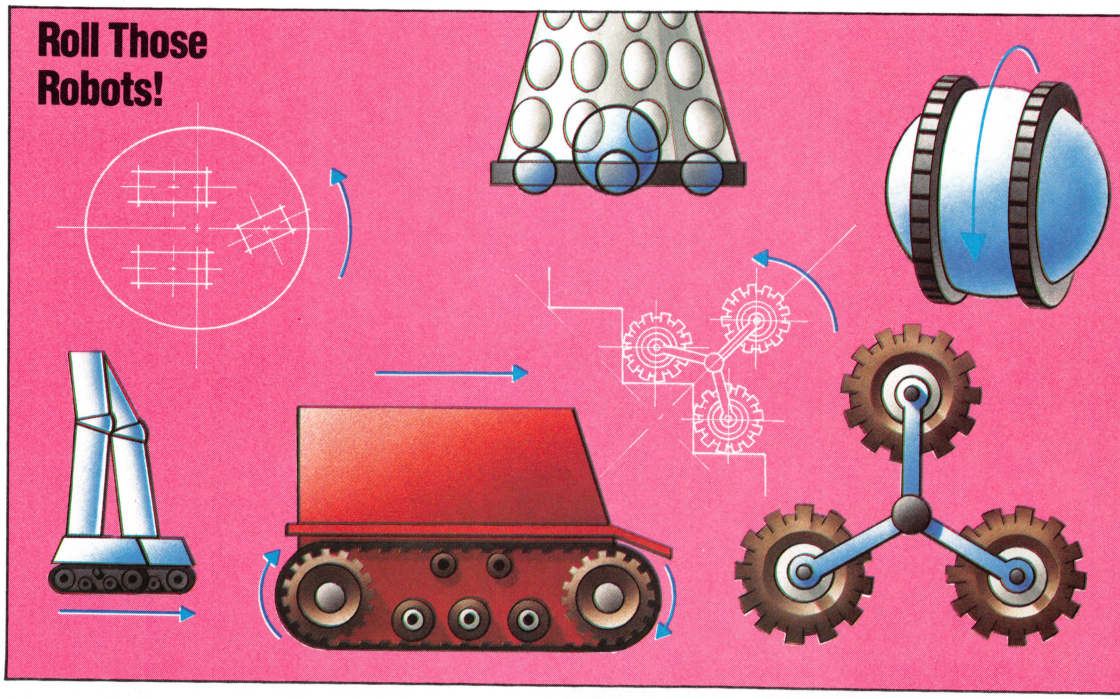
In the simplest electric motor, a flow of current in the rotor coil creates a magnetic flux opposed to that of the stator magnet's field; this opposition of forces causes the rotor to turn in the field



The stepper motor has many (sometimes hundreds) of coils. Switching current from one coil to another causes the whole to rotate in precisely controllable increments of arc

KEVIN JONES

Roll Those Robots!



Robots must move, and not just on smooth laboratory floors. We show a few of the possible mechanisms.

Tracked feet give grip at the cost of steerability, but allow a shuffling gait without leg-lift, thus lessening balance problems. Tracked robot vehicles are commonplace – with bomb squads and planetary exploration teams, for example.

The tri-axle format is the only wheel adaptation that allows the robot to climb steps.

A large roller-ball surrounded by ball-bearing stabilisers is very easily steered, but is sensitive to irregular surfaces. The trolley arrangement of two fixed wheels and a steered castor is the minimum necessary for stability.

Slinging the load inside large driving wheels is an attractive idea, but it raises the centre of gravity and lessens stability

KEVIN JONES

DIVIDE AND RULE

In this instalment of our LOGO course, we introduce a number of turtle graphics procedures that produce interesting shapes by the use of recursion. Some of these shapes have very strange properties indeed, as they were the inventions of mathematicians intent on demonstrating geometric paradoxes.

Our first program is designed to draw 'tree' shapes. To start with, we can simply draw a trunk with a right and a left branch. The branches may then be formed in exactly the same way (although they will be smaller), with a central main stem and right and left twigs. If this process is continued, a tree shape will gradually be built up. This is a good example of the way recursion can be used in LOGO.

Our procedure for drawing such a 'binary' tree requires two inputs: one for the length of the trunk and the other the 'level' number. The length of the branches is halved at each level away from the trunk.

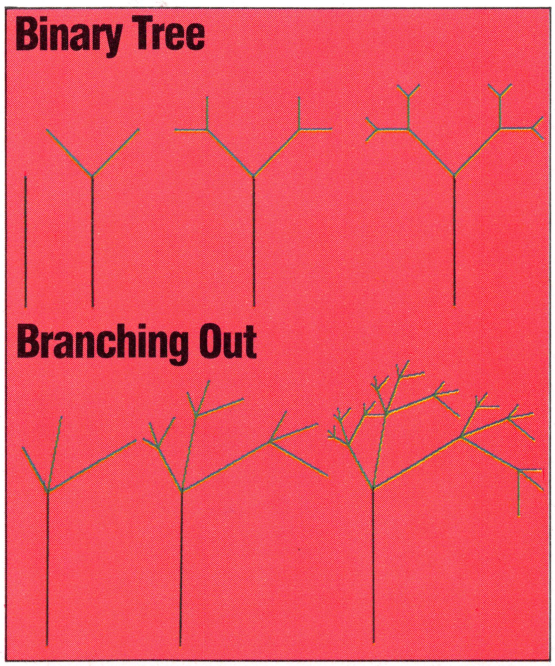
```
TO BRANCH :LENGTH :LEVEL
  IF :LEVEL = 0 THEN STOP
  FD :LENGTH
  LT 45
  BRANCH ( :LENGTH / 2 ) ( :LEVEL - 1)
  RT 90
  BRANCH ( :LENGTH / 2 ) ( :LEVEL - 1 )
  LT 45
  BK :LENGTH
END
```

Notice that the procedure is 'state transparent'. This is important as otherwise the 'state' of the turtle (its position and heading) would be changed each time the procedure calls itself, making it impossible to continue the drawing.

It must be admitted that this procedure produces an unrealistic tree — to make it more interesting, the procedure can be modified in various ways. Here's a version that draws three branches, each of a different length, at each level:

```
TO BRANCH1 :LENGTH :LEVEL
  IF :LEVEL = 0 THEN STOP
  FD :LENGTH
  LT 30
  BRANCH1 ( :LENGTH / 3 ) ( :LEVEL - 1 )
  RT 40
  BRANCH1 ( :LENGTH / 2 ) ( :LEVEL - 1 )
  RT 50
  BRANCH1 ( :LENGTH / 1.5 ) ( :LEVEL - 1 )
  LT 60
  BK :LENGTH
END
```

Try other modifications to produce more lifelike trees.



CHEQUERED POLYGONS

The following procedure draws a square, divides it into four, then divides each part into four, and so on:

```
TO BOARD :LENGTH :LEVEL
  IF :LEVEL = 0 THEN REPEAT 4 [FD :LENGTH RT 90]
    STOP
  BOARD ( :LENGTH / 2 ) ( :LEVEL - 1)
  FD ( :LENGTH / 2 )
  BOARD ( :LENGTH / 2 ) ( :LEVEL - 1 )
  RT 90
  FD ( :LENGTH / 2 )
  LT 90
  BOARD ( :LENGTH / 2 ) ( :LEVEL - 1)
  BK ( :LENGTH / 2 )
  BOARD ( :LENGTH / 2 ) ( :LEVEL - 1 )
  LT 90
  FD ( :LENGTH / 2 )
  RT 90
END
```

Write a similar procedure that splits a triangle up into four smaller triangles, then splits each of these up into four, and so on.

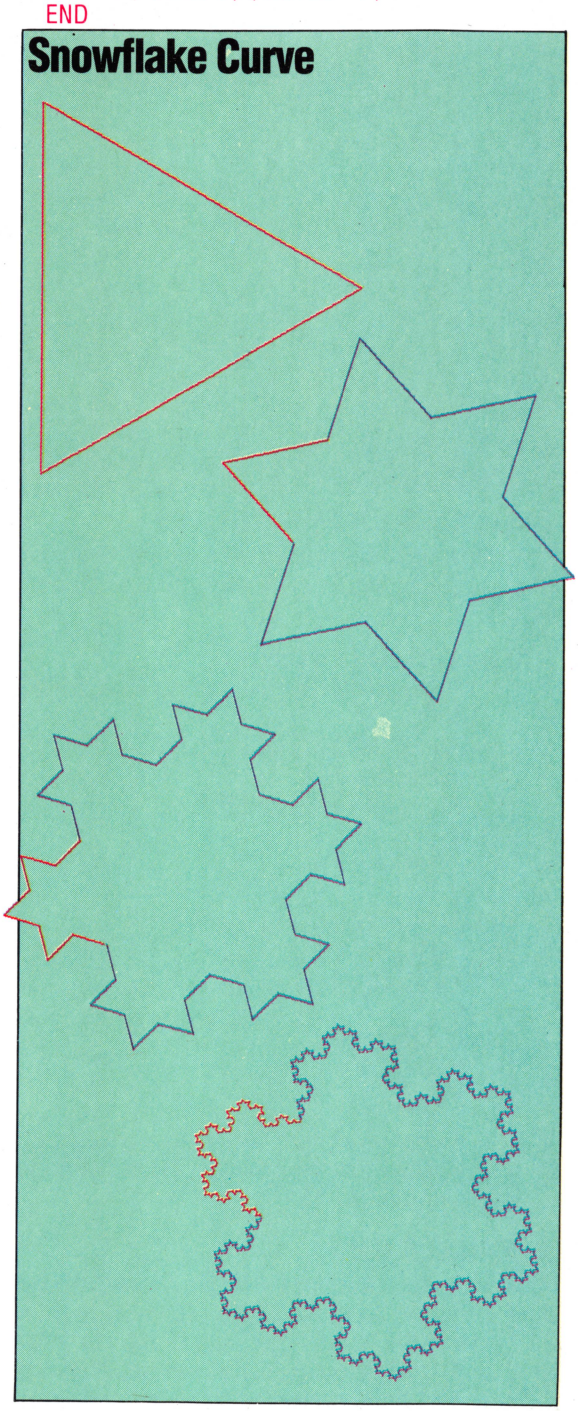
SNOWFLAKES

First draw an equilateral triangle — one whose sides are all of the same length. Now divide each side into three equal parts and draw a new equilateral triangle on the centre section. Rub out the common lines, then repeat this sequence for

each side of the new shape, and continue the process. The resultant shape is often called the 'snowflake curve' because of its appearance.

```
TO SNOW :SIZE :LEVEL
   REPEAT 3 [SIDE :SIZE :LEVEL RT 120]
END

TO SIDE :SIZE :LEVEL
   IF :LEVEL = 0 THEN FD :SIZE STOP
   SIDE ( :SIZE / 3 ) ( :LEVEL - 1 )
   LT 60
   SIDE ( :SIZE / 3 ) ( :LEVEL - 1 )
   RT 120
   SIDE ( :SIZE / 3 ) ( :LEVEL - 1 )
   LT 60
   SIDE ( :SIZE / 3 ) ( :LEVEL - 1 )
END
```



Notice that SIDE is not state transparent, but instead has been constructed so as to leave the turtle in the correct place for drawing the next side.

If this process of division is continued indefinitely (mathematicians use the phrase 'in the limit'), the result is a curve that has infinite length and yet surrounds a finite (fixed) area! It is possible to prove that this curve is neither one-dimensional nor two-dimensional, but is instead somewhere between the two.

A similar curve may be built up by starting with a square, dividing each side into three equal parts, constructing squares on the middle sections, and so on. Try writing a procedure that does this.

SPACE-FILLING CURVES

The series of curves shown here was invented by a mathematician called Sierpinski. If the process is continued in the limit, the result is a curve (a one-dimensional line) that passes through every point of the surrounding square (a two-dimensional shape). There are many other 'space-filling curves' that exhibit this strange behaviour.

The procedure used for drawing this curve is fairly complex. The level 1 curve is made up of four sides (shown in blue) that are joined by four diagonals (shown in red). So the main procedure, SIERP, just divides the process into four sections for the procedure ONE.SIDE to handle one at a time.

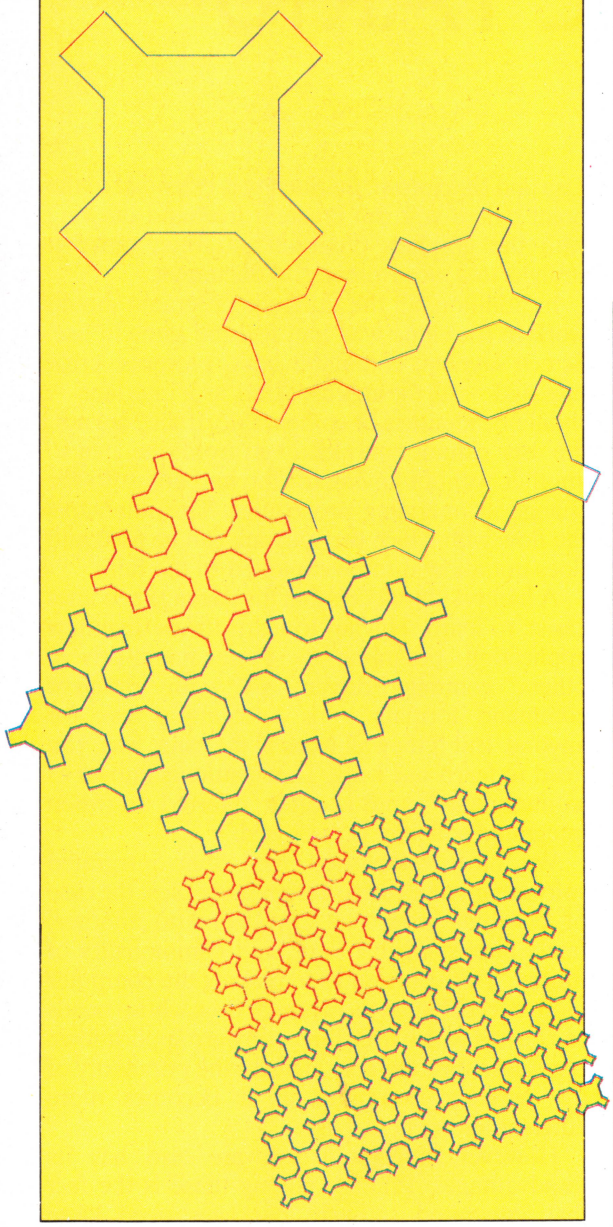
Consider just one of the sides. This is made up of three lines — a diagonal, a horizontal or vertical line, then another diagonal. At level 2, each diagonal is replaced by another, smaller, set of three lines, and the horizontal or vertical line is replaced by two similar sets of three joined by a line. The same process is carried out to move from level to level.

Here are the procedures for drawing the curves. Notice how the LOGO command MAKE is used to initialise DIAG:

```
TO SIERP :SIDE :LEVEL
   MAKE "DIAG :SIDE / SQRT ( 2 )
   REPEAT 4 [ONE.SIDE :LEVEL RT 45 FD :DIAG RT 45]
END

TO ONE.SIDE :LEVEL
   IF :LEVEL = 0 STOP
ONE.SIDE ( :LEVEL - 1 )
   RT 45
   FD :DIAG
   RT 45
   ONE.SIDE ( :LEVEL - 1 )
   LT 90
   FD :SIDE
   LT 90
   FD:SIDE
   LT 90
   ONE.SIDE ( :LEVEL - 1 )
   RT 45
   FD :DIAG
   RT 45
   ONE.SIDE ( :LEVEL - 1 )
END
```

Sierpinski's Curve



Logo Flavours

LCSI LOGO versions use SETPOS to set the turtle's position. This requires a list as an input, so the two co-ordinates must be combined with the command LIST. For example:

```
SETPOS LIST 45 67
```

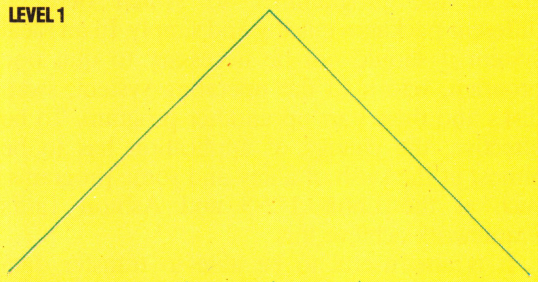
Remember also that LCSI versions use a different syntax with IF. A typical stop rule might be:

```
IF LEVEL = 0 [STOP]
```

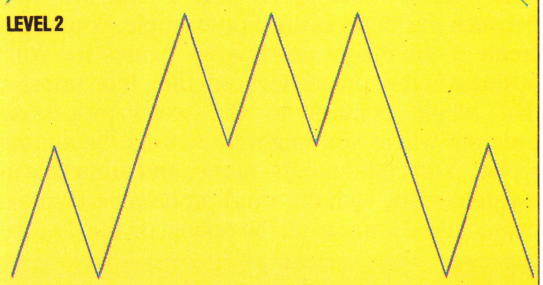
Procedure Problems 4

Our illustration shows a series of shapes that in their limit define a curve that has no gradient at any point. The first level consists of two lines — one going up, the other down. To proceed to the next level, we replace the ascending line by a broken line with six parts. This rises to half the height of the original line, then drops all the way down; it again rises to half the height, continues to full height, drops back to half height and finally rises to full height again. The descending line is divided into six sections in a similar way.

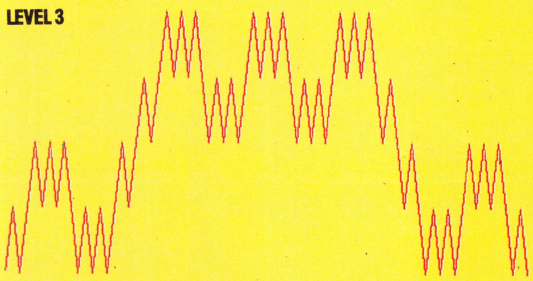
LEVEL 1



LEVEL 2



LEVEL 3



Try writing a set of procedures that will draw this series of curves. You should use SETXY instead of FD and RT. Your top-level procedure should divide the task into two parts — one concerned with going up, the other with coming down. You will then need to write two separate procedures to deal with the two parts of the top-level procedure. Remember that these procedures may call each other as well as themselves!

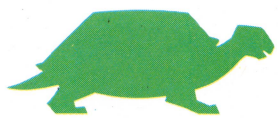
Exercise Answers

A recursive procedure that draws a tower of squares:

```
TO TOWER :SIZE
   IF :SIZE < 5 THEN STOP
   SQUARE :SIZE
   MOVE :SIZE
   TOWER ( :SIZE / 2)
END

TO SQUARE :SIZE
   REPEAT 4 [FD :SIZE RT 90]
END

TO MOVE :SIZE
   FD :SIZE
   RT 90
   FD ( :SIZE / 4 )
   LT 90
END
```



SUM OF THE PARTS

'Integrated software' has become one of the fashionable expressions of the software business. In this article we discuss exactly what is meant by integration and look at the advantages and disadvantages such a system offers. In future articles in this series we will look at specific packages.

Integration represents one of the most exciting trends in software ever. And while for the moment it applies mainly to business systems, its techniques have begun to filter down to home micros. An example of this is the Sinclair QL, whose four software packages encompass the main principles of integration (see page 502).

The main achievement of integration is to enable the programmer to switch between different packages quickly and simply. In an ideal system it should not be necessary to quit one program, return to the operating system, swap disks and then start up another program. To be effective, the change of application has to be almost at the push of a key and some programs, such as the Lotus 1-2-3 and Ashton Tate's Framework achieve this.

It is also useful to be able to transfer data between packages easily. For example, you might create a column of yearly sales figures for your business in the spreadsheet program, then transfer that whole column to the word processing program where you might be writing the annual report. You could use the names and addresses in a database file with the word processor to write a personalised letter to all the people on file. On the Lisa and Macintosh, this facility is extended to the point where you can create a freehand drawing with the graphics program and then move it straight to a word-processed document.

In addition, all the different programs should work in the same way and feel the same in use. Screen layouts, command keys, prompts, error messages — all the aspects of the 'user interface' — should be identical or comparable. If they are not, the user cannot confidently go from one area to another without having to stop and adjust to the change in operating procedures. This interrupts the flow with which the software can be used and does not allow it to be exploited to the full.

A handy side effect is that the package becomes easier to learn. Having to learn to use five new application programs — some menu-driven, others command-driven, all with different command formats — is a daunting task for anyone. But if they all work in the same way, you need to learn one only. This feature is known as 'commonality' and is often referred to when integrated software is discussed.

We have established then that integrated software involves three design principles: the ease of switching from one application to another; freely interchangable data; and commonality of format. This contributes to making the computer more accessible to the average user whose needs can be met with two or three software applications. It will also undoubtedly increase the popularity of the personal computer as it becomes more efficient and easier to use.

However, integrated software also has its disadvantages. Primary among these is the fact that integrated software packages need large amounts of RAM to operate. Imagine trying to fit a word processor, spreadsheet, and database — the three applications that are most commonly integrated — into 16 or 32 Kbytes. It can probably be done, but there would not be much, if any, room left to store data. It is this problem that restricts integrated software to machines with large memories: in general, to computers with 128 Kbytes or more. Of course, programs that are integrated can share some routines, so disk storage operations and other housekeeping activities need only to be written once. Nevertheless, each application has its own special requirements, and these take up space in RAM.

A second weakness of integrated software is an offshoot of the same problem of storage. To save on the amount of memory a program requires, software writers take shortcuts with the individual applications. A word processor that is built into an integrated package with two or three other

COURTESY OF MICROSCOPE

IAN McKINNELL

To Be Reviewed . . .
Among the integrated packages to be reviewed in this series are Lotus 1-2-3, Open Access, Symphony and Framework

applications cannot be as thorough and as complete as a word processor that stands on its own. The main reason for this is that a stand-alone program can take up as much memory by itself as the integrated programs take together.

One example can be found with two programs that run on the IBM PC and similar computers. Multimate is a program designed around the software that is used in Wang dedicated word processing equipment. It has many options for creating and formatting text that are not found in smaller programs, which makes creating very large and complex documents fairly simple. Multimate by itself requires 192 Kbytes of RAM to operate. Lotus 1-2-3, an integrated word processor, spreadsheet and database program, also requires 192 Kbytes. Yet the same space used by Multimate for one application must now handle all the software required for three complete programs in Lotus. As a result, the word processor in Lotus 1-2-3 is limited to simple memo writing.

A third disadvantage of integration also arises from one of its strengths. It is important, as we have seen, for integrated programs to look alike, so they are easy to learn and operate. Unfortunately, certain compromises have to be made by the software writers for this to be possible. The best way to operate a spreadsheet might not be the best way to use a database or word processor, so elements of the optimum design for each tend to be blended into a usable mixture. Microsoft found this to be a problem when designing a stand-alone word processor, Microsoft Word. The company wanted the screen display and the program's operation to be compatible with their hugely successful spreadsheet, Multiplan, so it would be easy to integrate the two programs. Microsoft included the same on-screen menu in Word that Multiplan users found so helpful, only to discover that the writers who needed a program like Word disliked having a menu on the screen all the time.

The point to remember about all software is that it has to do what the user wants. If a person has several tasks to do, like letter writing, simple accounts, and mailing lists, integrated software can make the job much easier. But there are sacrifices, and someone who wants to write a novel or very lengthy company reports on his microcomputer may have to continue to use separate, stand-alone word processors, spreadsheets, and database programs. Nevertheless, as software writers learn more about compressing computer instructions into smaller and smaller spaces in memory, and the memory in home machines starts to inch its way upward, integrated software will become more and more important to the home user as well as the business user. As a hint of things to come, two under-£500 computers are being sold with integrated software: Sinclair's QL, and the Commodore Plus/4.

In future instalments of this series, we will take a closer look at some of the integrated programs that are having a large effect on software development.

We will examine two distinct approaches to integration: one exemplified by Lotus 1-2-3 and similar programs, in which the applications, although working together, look very much like typical computer programs have always looked and the second type found on machines like Lisa and Macintosh, where the whole operating environment is designed for integration.

FLOPPY DISK

In considering the development of computing in the post-war period it's easy to assume that the techniques of Large-Scale and Very Large-Scale Integration (LSI and VLSI) have been the fuel, and the cheap mass-produced microprocessor chip the engine, of the microcomputer revolution. Of equal importance, however, has been the development of cheap, fast, dependable back-up storage — the *floppy disk*. Single-drive units costing about the same as a microcomputer that are capable of storing up to half a Megabyte of data or programs on a disk costing a couple of pounds make an over-priced electronic toy into a credible data processing system.

Inevitably in the growth of microcomputing, several disk formats have developed: the 8in diameter floppy, the 5¼in mini-floppy, and the 3in and 3½in microfloppy. All consist of an oxide-coated thin (less than 0.5mm) flexible plastic disk, in a protective jacket. This jacket is never removed, and is pierced with a radial slot or window that gives access to the disk surface. In use the disk spins inside the jacket at around 300 rpm, while the drive read-write head moves backwards and forwards in the window. This allows reasonably fast access to every spot on the surface. Another slot in the jacket is called the *write-protect* notch: the drive unit checks that this notch is open before writing on the disk.

The read-write head is similar to a tape-recorder's; it writes by changing the alignment of the *magnetic domains* in the disk's surface, and reads by sensing those alignments. Information is written along concentric circles on the disk surface called *tracks*, divided into 20 to 40 *sectors*. High-quality, or *double-density*, disks can have up to 80 tracks, while *single-density* disks have 40. The boundaries between track sectors can be marked by photoelectrically-sensed holes — one per sector — in the inside rim of the disk, in which case the disk is *hard-sectored*. The alternative — soft-sectored — disks have just one index hole, which marks the start of the first sector on every track, subsequent sector boundaries being magnetically marked.

DAVID WEEKS

Demo Disk
The illustration shows a 5¼in floppy disk, with the read-write window, the sector index hole and the write-protect notch clearly visible

FLOWCHART

Any graphical method of representing the interactions of control or information in a program or system tends to be called a *flowchart*, though many such diagrams represent relationships rather than flows, and so should really be called 'process diagrams', or 'data graphs'. The commonest form of flowchart comprises boxes of various shapes representing program processes such as input/output, decision-making and data processing; these boxes are linked by arrowed lines that show how control passes from one process to another in the program. Most self-taught programmers use this type of flowchart, but in recent years it has been discarded by professionals and academics because it allegedly distorts or conceals program structure when used as descriptive documentation, and because it encourages badly structured program design. In general, the graphic approach is more and more discarded in favour of precise requirements specifications, data definition statements and function description languages.

FLOW CONTROL

In any real-time data communications application the receiver and the sender are likely to have different optimum transmit/receive rates; in order that slow receiving devices are not overwhelmed by fast transmissions, some *flow control* strategy must be employed, either by the transmitter, or by the transmission network controller.The simplest strategy is known as *end-to-end* control, in which the amount of data sent is limited to the capacity of the receiver. Another strategy is called *hop-by-hop* — the amount of data sent is limited at every step through the transmission network by the capacity of the node or link carrying the information.

FORMAT

Any pre-defined structure is a *format*, but in computing the term usually refers to disk or instruction formats. The former refers to the way in which the pattern of tracks and sectors is physically distributed on the surface of a magnetic disk (see page 124). Disks used by one manufacturer's disk drives may well be unreadable to other drives precisely because of this. This problem is eased somewhat by the emergence of common operating systems such as CP/M and MS-DOS: software packages written for these systems often share a common disk format. To format a disk is to prepare it for a particular disk drive; this destroys any information on the disk.

Instruction formats describe the syntax of instructions or commands. For example:

```
RENUMBER startno [,endno [,inc] ]
```

indicates the spelling of the command word, the nature of its possible arguments (first line number, end line number and increment), the delimiters (the comma), and the variant forms of the command (anything in square brackets is optional).

CUDDLY TOY

For Commodore 64 artists the Koala-pad offers an easy way to produce high-quality graphic displays and overcomes the problems associated with graphics generation on this machine. Light, compact and easy to use, this peripheral device enables complex screen pictures to be built up at the touch of a finger.

Despite its ability to produce excellent graphic effects, the Commodore 64 has so far not been provided with the high-quality graphics peripherals that have been produced for the BBC Micro. This is probably a result of the difficulties associated with the production of high resolution graphics on the 64 – difficulties that have deterred manufacturers from producing such a device. Now, however, Audiogenic has begun importing a graphics pad made by the American company Koala Technology that allows Commodore 64 owners easier access to the machine's high resolution graphics capabilities.

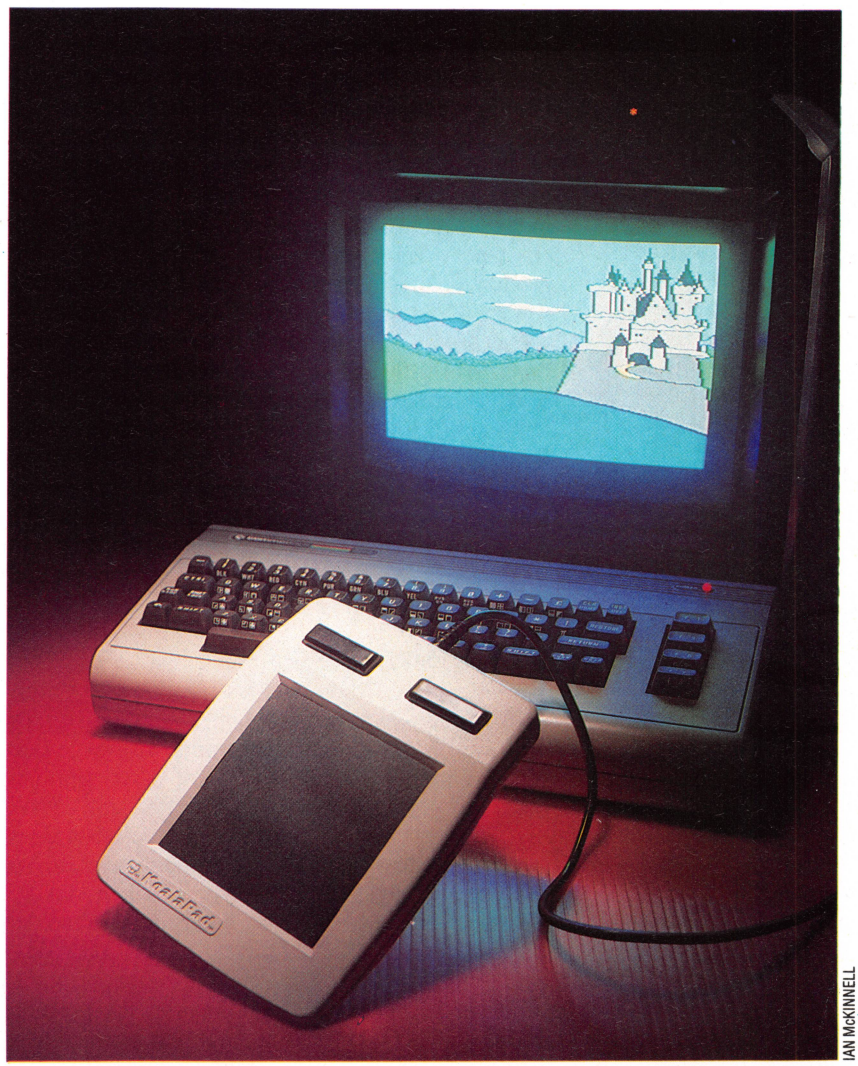
Unlike similar touch tablets such as the Grafpad (see page 169), the Koala-pad is light and compact, measuring only 20.5 by 16 cm (8 by 6in). In the centre is an 11 by 11 cm (4 by 4in) carbon fibre square that covers a touch-sensitive membrane that is similar to the Spectrum keyboard. By simply pressing a finger or a pen onto the membrane, the user can guide a cursor around the screen. This is in contrast to other graphics tablets, which require a special 'stylus' to complete the circuit.

The membrane consists of two sheets of conducting wires – one in the horizontal and one in the vertical axis. When the membrane is pressed, the pad detects which wires are in contact and sends the resulting co-ordinates to the computer. Above the touch membrane are two buttons, one of which must be pressed when the user wishes to colour in a point on the screen or select one of the various paint options that are available. Either button may be used – presumably this is to cater for both left- and right-hand users.

COLOURING IN

The Koala-pad connects to the computer via the joystick port, and Koala Painter (the software needed to work the pad) is loaded from disk. Once loaded, a display of the various options available appears on screen. At the bottom of the screen is the 'palette' containing 16 'true' and 16 'tinted' colours. The tints are built up by colouring alternate pixels in different hues, giving the effect of shading. Above the palette are eight boxes containing the 'brushes'. These simply consist of various shapes that may be plotted on the screen, and range from a single pixel to combinations of pixels and lines. Surrounding the brushes are the various options for drawing lines or shapes on the screen. These are selected by the user pressing on the pad membrane and thus directing a cursor arrow. When the arrow points to the desired option, pressing one of the Select buttons on the pad brings it into effect. The option flashes to remind the user which mode is currently in use. The Koala-pad provides the facilities to produce single lines, rays (lines drawn from a single point), frames and circles. Blocks of colour may be added by using the 'box' (coloured squares) or 'disc' option (coloured circles). Further colouring is achieved by using the FILL command to fill an enclosed area with a chosen colour. Colours are altered by using the X-COLOUR command.

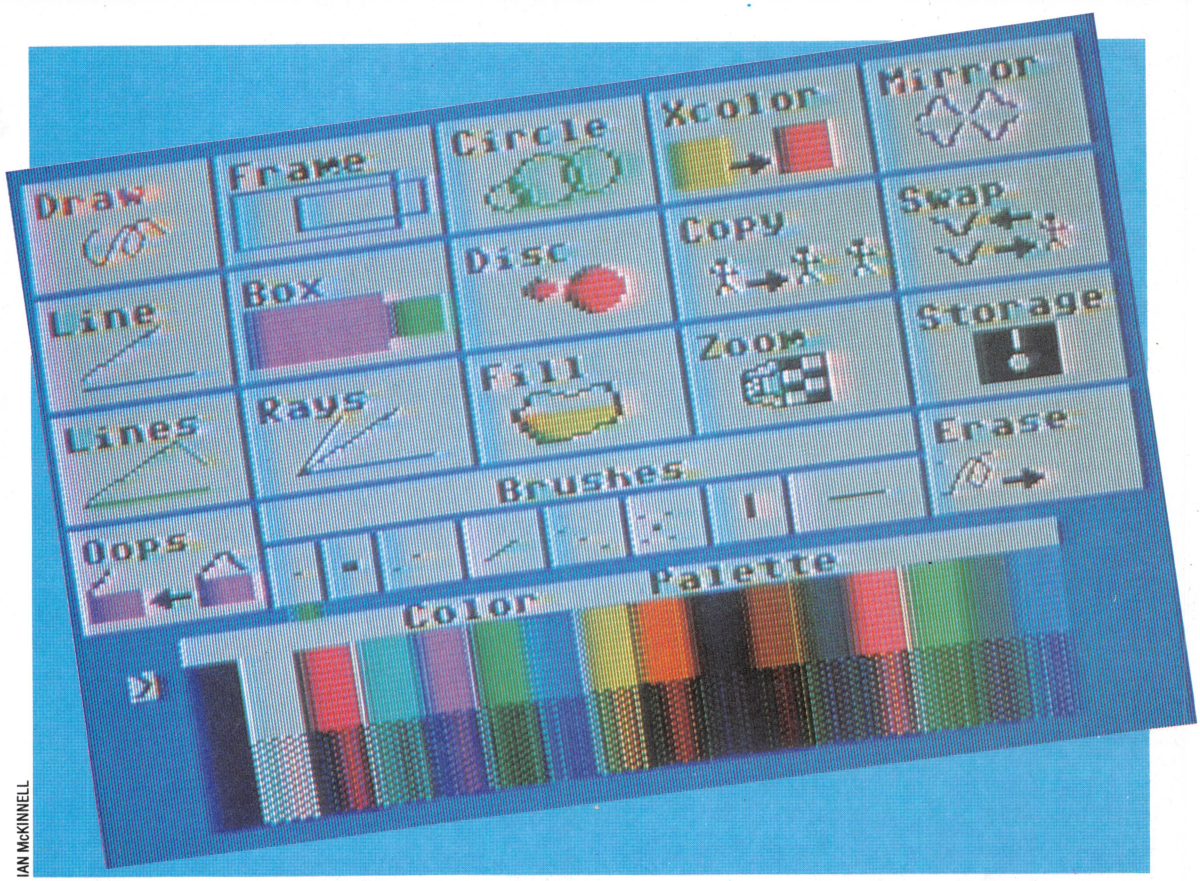
Identical figures may be drawn simultaneously by using the MIRROR command. This divides the screen into four sections, with the cursor restricted to the top left-hand quarter. Anything plotted within that



Hand-Held
Complex and sophisticated graphics screens can be built up in a few hours using the easily understandable menu-driven software. Unlike many similar graphics tablets, the Koala-pad can be held in the hand while being operated

IAN McKINNELL

Take Your Pick
The main menu of the Koala-pad consists of boxes containing both the name and an explanatory icon. Although the icons are intended to aid understanding, some of the illustrations are perhaps confusing. The cursor arrow is moved into a box and the 'Select' button is pressed. The name of the selected box will flash to remind the user which mode is in use



IAN McKINNELL

quadrant will automatically be copied to the corresponding area of the other three quarters.

Highly detailed pixel plotting is achieved by using the ZOOM command. The user may choose any part of the screen, and this is then displayed as an expanded 'window' at the bottom of the screen. The individual pixels are displayed as eight by eight pixel, character-sized squares. This feature makes the production of fonts and sprite-sized figures quick and simple. These figures are then placed anywhere on the screen by using the COPY command; this allows the user to define an area of the screen, the contents of which are copied to any other position.

After these options have been selected, the cursor arrow is moved off the screen and the Select button is pressed. The screen then changes to the 'canvas' on which the required picture is built up. By moving the cursor arrow and pressing the Select button, lines and shapes may be drawn anywhere on the normal Commodore 64 graphics screen.

MINOR NIGGLES

The quality of the graphics produced by this device is excellent, rivalling high resolution screens produced with commercial software. However, one disappointment – also shared by other graphics packages – is the quality of the freehand DRAW command. The membrane matrix resolution does not match that of the 64's high resolution screen, so the user's stylus (or fingertip) will often not be directly over a grid intersection and may in fact be triggering two points at the same time. The computer, in trying to interpret this, will plot a point, which unfortunately is not always at the position intended. This can result in what was planned as a straight line appearing as a messy scrawl. Another criticism is that, apart from the ZOOM command, there is no option to change the colour without returning to the main menu. But these are minor complaints that are more than offset by the speed at which the LINE and FILL commands are executed.

A further software limitation that could have been better thought out is the method used to erase mistakes. When a mistake is made, it is removed from the screen by use of the OOPS command, which is accessed from the main menu. However, using OOPS will erase all the work that has been performed since the user last exited from the main menu. This means that perhaps half an hour's work may be erased, simply because of a single error. The alternative is to erase a mistake by using the ZOOM command and correcting the error pixel by pixel, which in the case of a badly placed disc or box could take some time. A welcome amendment would be to restrict the extent of the OOPS command to the last press of the Select button rather than the last exit from the main menu.

As may be suspected from a device that plugs into the joystick port, it is possible to use the Koala-pad as a joystick, thus allowing users to access the Koala-pad from their own programs. The position of the cursor can be obtained from BASIC by PEEKing locations 54297 and 54298 for the co-ordinates of X and Y respectively.

Screens may be saved onto disk and can easily be transferred to the user's own programs, allowing the development of 'Hobbit'-style adventures with text at the bottom of the screen and a picture above. By using the Koala-pad it is possible to save and recall up to 16 different eight-Kbyte screens on a disk. Although it is not possible to load a screen from disk

directly into screen memory, Koala Technology has provided a program in the Koala Painter user guide that enables screens to be transferred from the area in which they are loaded to the screen memory.

The maximum resolution of a saved Koala Painter screen is 255 by 255 pixels if it is included in a BASIC program, although higher resolutions may be gained by using machine code. This limitation may cause problems as the Commodore 64 high resolution screen is actually 320 by 200 pixels in size. The Koala-pad is restricted to a maximum of 255 pixels because this is the largest number that can be addressed by one byte – to address a full screen would require 16-bit addresses. Unfortunately this may result in the user losing a portion of the display. This is because, when the screen is not actually being displayed, it will be stored somewhere within user RAM. Therefore the eight-Kbyte screen, together with the colour information, must be transferred from its location in user memory to the high resolution screen memory commencing at location 55296.

However, any minor niggles apart, it must be said that the Koala-pad is a very useful peripheral for anyone wishing to produce high resolution graphics screens on the Commodore 64. It is a pity that the software is supplied on disk only, as this will inevitably limit the device's appeal. But for disk owners, this would be a worthwhile investment – particularly for the artist or the adventure-writer.

KOALA-PAD

PRICE
£79.95

DIMENSIONS
210 × 165mm

SCREEN
The full 320 × 200 Commodore 64 screen display is used, though this may be restricted to 255 × 255 when calling a screen from a BASIC program

INTERFACE
Connects through the joystick port of the C64

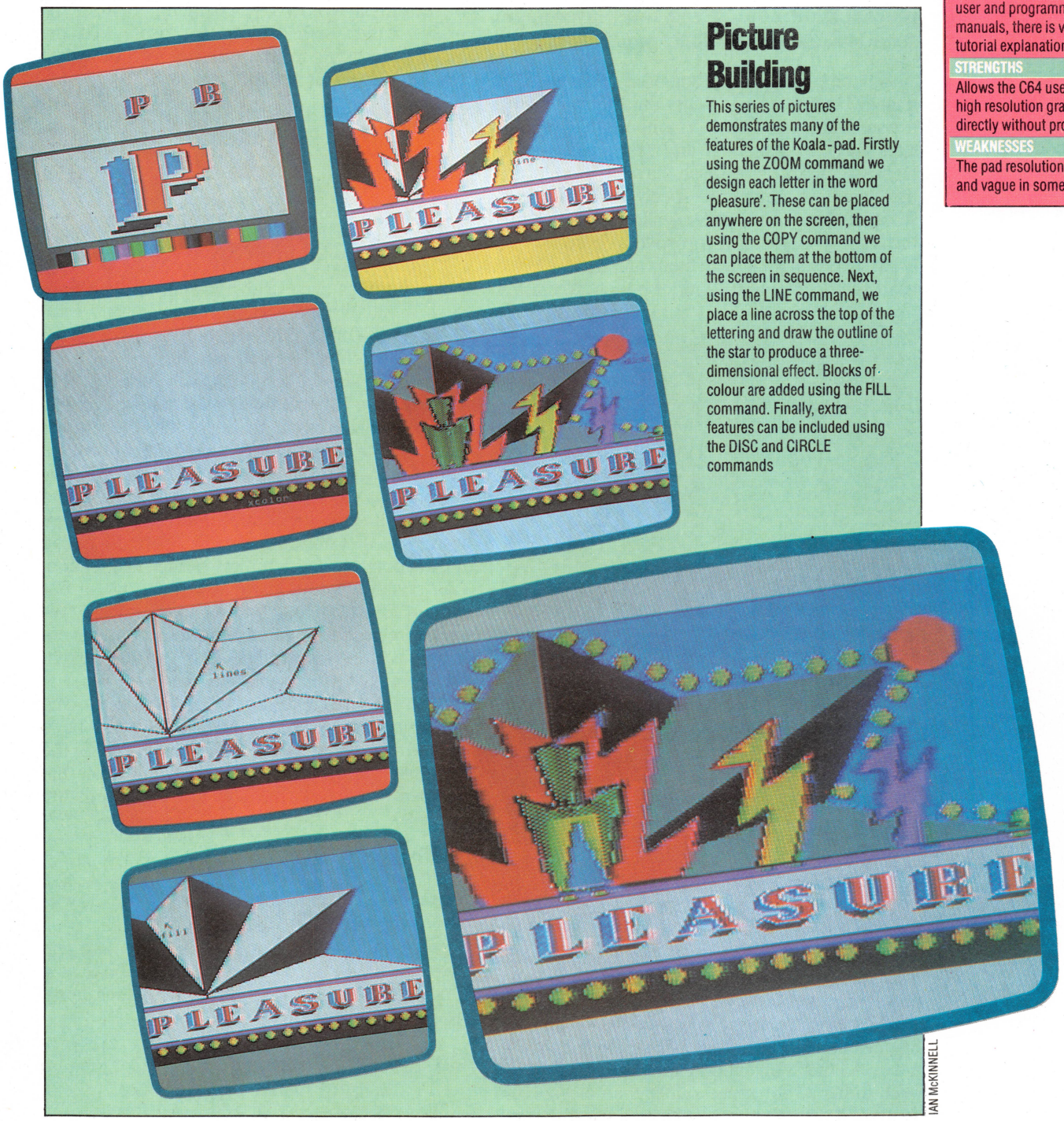
DOCUMENTATION
Although most of the information required is in the user and programmer's manuals, there is very little tutorial explanation

STRENGTHS
Allows the C64 user to produce high resolution graphic screens directly without programming

WEAKNESSES
The pad resolution is coarse and vague in some modes

Picture Building

This series of pictures demonstrates many of the features of the Koala-pad. Firstly using the ZOOM command we design each letter in the word 'pleasure'. These can be placed anywhere on the screen, then using the COPY command we can place them at the bottom of the screen in sequence. Next, using the LINE command, we place a line across the top of the lettering and draw the outline of the star to produce a three-dimensional effect. Blocks of colour are added using the FILL command. Finally, extra features can be included using the DISC and CIRCLE commands



IAN McKINNELL

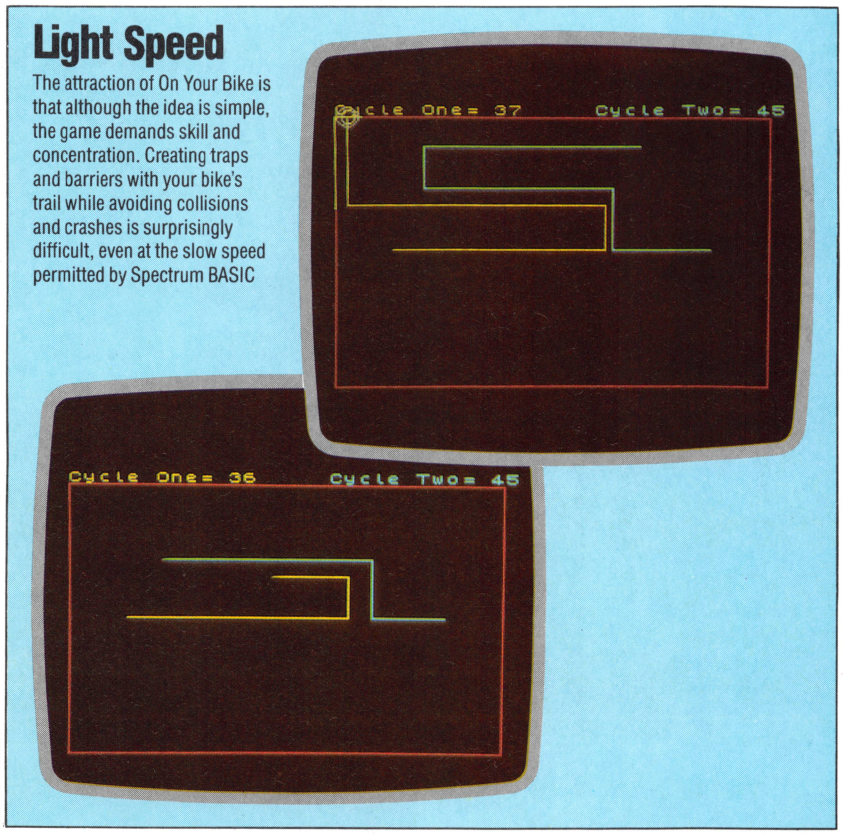
ON YOUR BIKE

Though most successful commercial games are fairly lengthy, and written in machine code for speed, it is possible to write an entertaining game in BASIC. The game that follows is fairly simple, and indeed takes up only 35 lines of BASIC, but is still good fun to play. Furthermore, it is a two-player game, so is less anti-social than most!

The game is called 'On Your Bike' and is based on a scene from the classic Walt Disney film *Tron*. It is a contest between two opponents that requires skill and fast reactions and takes place in an enclosed arena. You each have a bicycle that travels at an incredible speed and cannot be stopped. Your only control allows you to turn through 90 degrees at top speed.These bicycles leave solid walls of light in their trail, and the object of the game is to force your opponent to crash in the ever-tightening maze you create as you zoom around the arena.

Light Speed

The attraction of On Your Bike is that although the idea is simple, the game demands skill and concentration. Creating traps and barriers with your bike's trail while avoiding collisions and crashes is surprisingly difficult, even at the slow speed permitted by Spectrum BASIC



IAN McKINNELL

The game has been implemented on the ZX Spectrum, which is not known for the speed of its BASIC. As this is an action game, the program has been designed for speed rather than elegance,so much of the listing may seem a little unstructured. Subroutine calls, and other structured devices have been avoided as they would have sacrificed execution speed.

The first stage is the design of the arena and score display. As you can see this is fairly simple, which contributes to the shortness of the final program.The only point to note is that the border of the arena is now one character in from the usable screen area. This is to ensure that the graphics resulting from a collision with the arena wall do not go off screen:

```
10 LET p=0: LET q=0
100 BORDER 0: PAPER 0: CLS
110 PRINT AT 0,1; INK 6; "Bike One= "; q
120 PRINT AT 0,19; INK 5; "Bike Two= "; p
130 INK 2
140 PLOT 8,8: DRAW 239,0: DRAW 0,159
150 DRAW -239,0: DRAW 0,-159
```

The arena has been drawn in red, and we have chosen yellow to represent bike one, and cyan (blue) for bike two. The variables p and q hold the current score for the two contestants.

The next stage is to initialise all variables, and here we have to start thinking about how we are to implement the main action of the game. The action for a single bike is fairly straightforward, and is shown in the flowchart. Using POINT we check if the bike's current position is occupied, and move to the collision routine if it is. If it is not, we move into that position using PLOT, and then read the keyboard to check for any change in direction. Our position is then incremented by one in our current direction, and the cycle begins again. We therefore need four variables: two for our current x and y co-ordinates, and two for our current direction along the x and y axes.

However, we are dealing with two bikes moving at the same time. An elegant solution would be to use four two-element arrays, x(2) and y(2), for the positions for example, but this would slow the game down so we have to use eight separate variables:

```
200 LET x=40: LET y=88
210 LET m=215: LET n=88
220 LET a=1: LET b=0
230 LET i=-1: LET j=0
```

This sets the initial positions of the bikes, and sets them moving towards each other one pixel at a time. The basic action of the game is then fairly simple to implement:

```
400 IF POINT (x,y)=1 THEN LET col=6: GOTO 700
410 IF POINT (m,n)=1 THEN LET col=5: LET x=m:
    LET y=n: GOTO 700
420 PLOT INK 6;x,y:PLOT INK5;m,n
```

(lines 500 to 570 are the keyboard routine that sets new values for a, b, i and j)

```
600 LET x=x+a: LET y=y+b
610 LET m=m+i: LET n=n+j
620 GOTO 400
```

The only confusing point is perhaps line 410, where the variables for bike one are set to those for bike two, and a new variable,col, is introduced. This is so that a single routine can be used for the collision action, where x and y are simply used to indicate the point at which the collision takes place, and col sets the colour.

The routine for checking the keyboard has to be fast, but we regretfully have to use IF . . . THEN statements that are fairly slow. However, we can use the fast IN command to read the keys. The control keys chosen are Q and A, which control upward and downward movement for bike one, and P and ENTER for bike two. Left and right are X and C for bike one, and N and M for bike two. (See page 366 for a full explanation of how blocks of keys relate to bytes in memory.)

```
500 IF IN 64510= 190 THEN LET a=0: LET b=1
510 IF IN 65022= 190 THEN LET a=0: LET b= -1
520 IF IN 65278= 187 THEN LET a= -1: LET b=0
530 IF IN 65278= 183 THEN LET a=1: LET b=0
540 IF IN 57342= 190 THEN LET i=0: LET j=1
550 IF IN 49150= 190 THEN LET i=0: LET j= -1
560 IF IN 32766= 187 THEN LET i=1: LET j=0
570 IF IN 32766= 183 THEN LET i= -1:LET j=0
```

All that remains is the collision routine, and the updating of the scores. An expanding series of concentric circles, centred on the point of impact was chosen, with radii of four, six and eight pixels:

```
700 FOR d=1 TO 3
710 CIRCLE BRIGHT 1; INK col; x,y,2+d*2
720 NEXT d
730 IF col=6 THEN LET p=p+1: GOTO 750
740 LET q=q+1
750 GOTO 100
```

This finishes the game, with the last statement looping back to the initialising procedures at the beginning. The game could, however, do with a starting procedure to make it more friendly to use:

```
300 PRINT AT 10,5; INK 7; "PRESS ANY KEY TO
    START"
310 IF INKEY$= "" THEN GOTO 310
320 PRINT AT 10,5; "
                   "
```

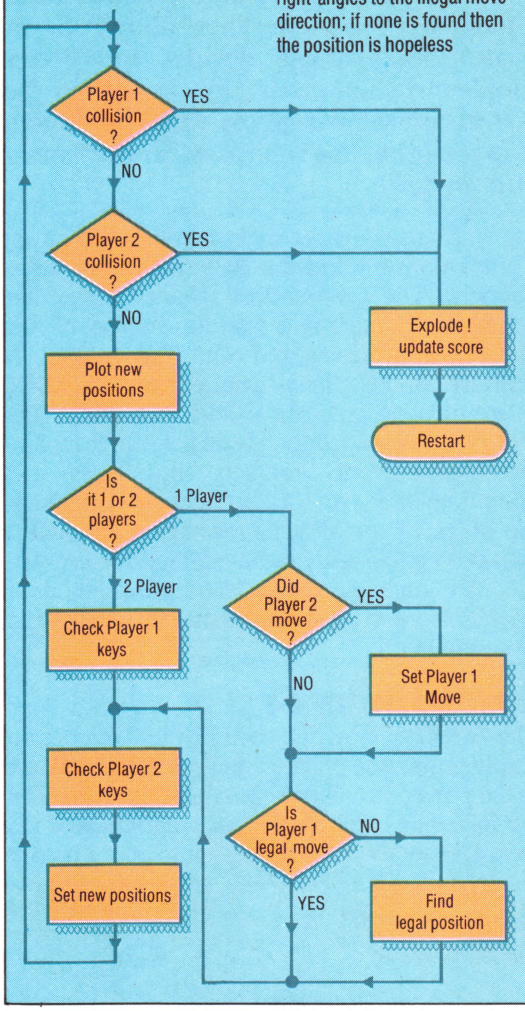
This gives you a break between consecutive rounds. All that remains is to save the game to cassette, preferably using SAVE "OnYerBike" LINE 10 so that the game automatically runs as soon as it is loaded.

The game could obviously be made more exciting, with instruction screens, loading screens, a one-player option with a strategy routine controlling the other bike as we have suggested, sound and better graphics. But certainly the latter options would make the game unplayably slow. In a future instalment we will rewrite this game in machine code to demonstrate its full potental.

One Hand Clapping

Fitting In

In the one-player version the computer is Player 1. The program section that checks Player 1's command keys for input is bypassed, and the algorithm in the code given allows Player 1 to continue moving in a straight line until that generates an illegal move, or Player 2 moves. In the latter case, this move is mirrored or duplicated, and then checked for legality. Whenever an illegal Player 1 move is generated, the program looks for legal moves at right-angles to the illegal move direction; if none is found then the position is hopeless

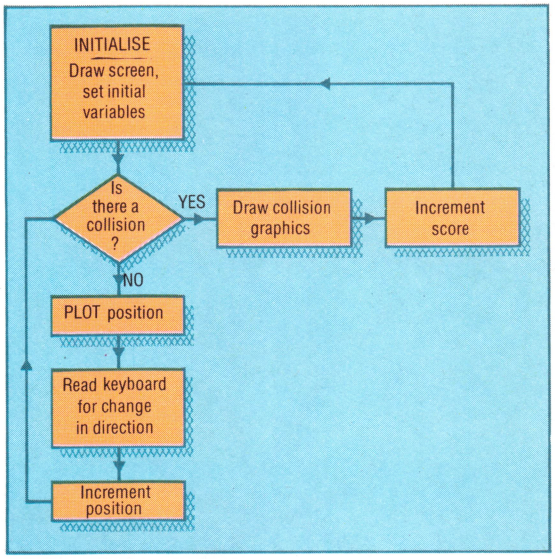


There are many ways of implementing a one-player version of this game. The changes we suggest make it possible to choose between the one- and two-player versions at the start of every game. It isn't difficult to invent satisfactory algorithms for playing this game, but it's extremely difficult to implement them in BASIC without slowing down the game considerably. Make the following changes to the program:

```
20 LET keybd=500:LET
   pt=-1:LET umoved=0:
   RANDOMIZE
260 PRINT AT 10,5;"No. of
   Players (1/2) ?"
270 LET a$=INKEY$:IF
   a$<>"1" AND a$<>"2"
   THEN GO TO 270
280 IF INKEY$<>"2" THEN
   LET keybd=440
```

These lines give the user the choice of game types, and implement that choice by accessing either the one-player stratagem between lines 440 and 460, or the standard two-player version between lines 500 and 570. Our strategy is contained in these lines:

```
430 GOTO keybd
440 LET pt=SGN (RND-0.5): IF
   umoved=pt THEN LET a=pt*j:
   LET b=pt*i: LET umoved=0
450 IF POINT (x+a,y+b)<>1
   THEN GOTO 540
460 LET pt=SGN (RND-.5): LET
   a=a*pt: LET b=b*pt: LET d=a:
   LET a=b: LET b=d: IF POINT
   (x+a,y+b)<>1 THEN GOTO
   540
490 LET a=-a: LET b=-b: GOTO
   540
540 IF IN 57342=190 THEN LET
i=0: LET j=1: LET umoved=1
and similarly add:
: LET umoved=1
to the end of lines 550 to 570
```



LIZ DIXON

One-Player Option

This simplified flowchart shows the program structure with only one player. Each process is repeated in the full two-player game

TRAFFIC CONTROL

So far in this series we have constructed an input/output system that allows us to control low voltage devices and accept simple switched inputs. We have used this box to control a Lego car with two motors. Now we bring this vehicle under the control of a joystick.

The ways in which joysticks can be used on the Commodore 64 and the BBC Micro are rather different. The Commodore 64 uses a standard Atari-type joystick that operates by way of four directional switches and one fire button. In contrast to this digital arrangement, the BBC Micro uses an analogue joystick, or paddle. This type of joystick does not rely on the simple making or breaking of contacts but uses two potentiometers, one for left/right movement and the other for up/down movement (although a digital-type joystick can be adapted for use with the BBC analogue port). As these methods of operation are so different, we shall talk about each type separately.

COMMODORE 64

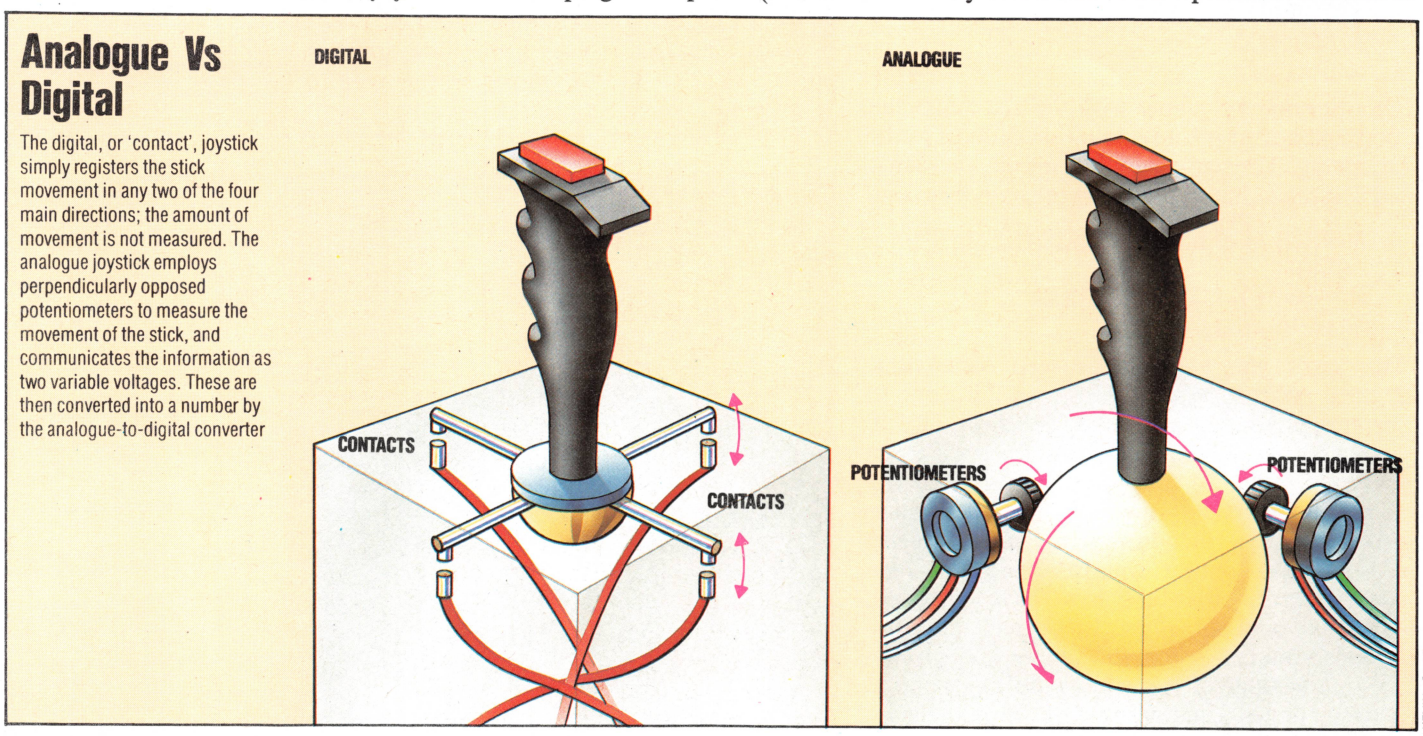
The Atari-type joystick used by the Commodore 64 plugs into one of the games ports, located next to the power on/off switch. We shall use port 2 in the following explanation and program, so if you have a joystick available plug it into port 2 (nearer the on/off switch). If the joystick is moved from the central position towards the top, one of the four internal switches will be closed. Port 2 links directly into memory location 56320 and closing this switch will cause one of the bits in this location to go low, rather in the same way as closing an external switch connected to the buffer box will cause a bit in the user port data register to go low. Type in the following short program that repeatedly displays the value of the data register. With the program running, move the joystick around and press the fire button, noting the various changes that take place in the value displayed on the screen.

```
10 REM **** CBM 64 READ JOYSTICK ****
20 PORT2=56320
30 JOY=PEEK(PORT2):GOSUB500
40 PRINT CHR$(145);JOY,B$
50 GOTO30
60:
500 REM CONVERT TO BINARY S/R
510 B$="":N=JOY
520 FOR D=1 TO 8
530 N1=INT(N/2):R=N-2*N1
540 B$=B$+STR$(R):N=N1
550 NEXT D
560 RETURN
```

After a few minutes experimentation it should become clear which bits in the joystick location correspond to the four direction switches and fire button. Normally, with the joystick in the central position, the contents of the joystick location are 127, i.e. 01111111. Pushing the joystick causes the value to change to 126 (01111110). Bit 0 is obviously connected to the up direction switch in

Analogue Vs Digital

The digital, or 'contact', joystick simply registers the stick movement in any two of the four main directions; the amount of movement is not measured. The analogue joystick employs perpendicularly opposed potentiometers to measure the movement of the stick, and communicates the information as two variable voltages. These are then converted into a number by the analogue-to-digital converter



the joystick. We can summarise the actions of the joystick's switches on the contents of the joystick memory location as follows:

Joystick	Decimal	Binary
Central	127	01111111
Up	126	01111110
Down	125	01111101
Left	123	01111011
Right	119	01110111
Fire	111	01101111

You may also have noticed that moving the joystick diagonally can cause two switches to close simultaneously. Although we do not require detection of diagonal movement to control our vehicle, the results of such movements on the joystick memory location are as follows:

Joystick	Decimal	Binary
Up and left	122	01111010
Up and right	118	01110110
Down and left	121	01111001
Down and right	117	01110101

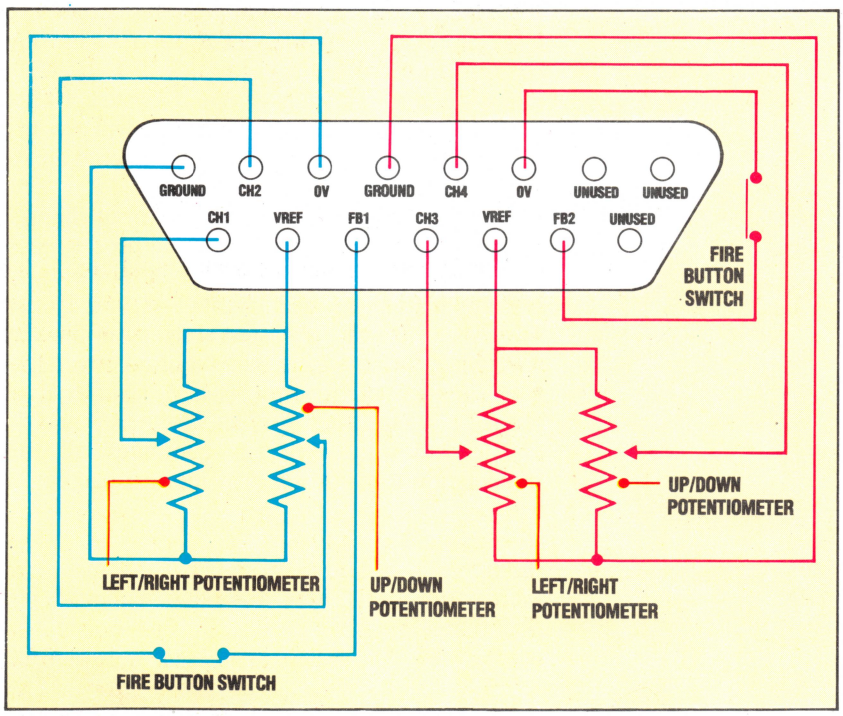
The following program uses a joystick to control the movements of the twin motor vehicle. The vehicle should be connected to the output box in the same way as on page 586 and the joystick should be plugged into games port 2, located on the right-hand side of the Commodore 64.

```
10 REM CBM 64 JOYSTICK
20 DDR=56579:DATREG=56577
30 POKEDDR,255:REM ALL OUTPUT
40 JOY=PEEK(56320):REM JOY PORT 2
50 GOSUB1000:REM TEST JOYSTICK
60 POKEDATREG,0:GOTO40
90 :
1000 REM TEST JOYSTICK S/R
1005 IFJOY=127THEN POKEDATREG,0
1010 IFJOY=126THEN POKEDATREG,5
1020 IFJOY=125THEN POKEDATREG,10
1030 IFJOY=123THEN POKEDATREG,6
1040 IFJOY=119THEN POKEDATREG,9
1050 IFJOY=111THEN POKEDATREG,0:END
1060 RETURN
```

BBC MICRO

The BBC joystick is an analogue device relying on two potentiometers to provide information about the up/down and left/right movements made. The essential difference between a digital joystick and the analogue type used by the BBC Micro is that the latter gives information about 'position' within given limits, whereas the former gives information about direction of travel only. The joystick potentiometers work in the following manner.

A potentiometer is essentially a resistance across which a voltage is applied. A third connector to the potentiometer can move along the resistance taking a fraction of the supply voltage. This fraction depends on the position of the moving connection. On a linear type of potentiometer, if the moving connection were positioned halfway across the resistance, then the voltages tapped off would be half the supply voltage. Thus, by moving the central connection, any voltage between zero and the supply voltage can be obtained. Turning the volume control on a radio or record player is essentially moving the middle connection across the resistance of the volume control potentiometer. In an analogue joystick the movement of the middle connection is made by moving the joystick handle.



BBC Micro joysticks are generally supplied in pairs. The BBC Micro's analogue port connections for joysticks 1 and 2 are shown in the above diagram.

A reference voltage is provided by the micro across each potentiometer and the middle connection's tap-off voltage is input through two channel inputs. Channel 1 is used for the input from the left/right potentiometer and channel 2 is used to accept input from the up/down potentiometer. The fire button is a simple make or break switch.

Once the potentiometer inputs have been accepted they must be converted from analogue to digital form by an internal converter. This conversion is done by comparing the input voltage with the reference voltage and conversion time is around 10 milliseconds for each channel read. Once the joystick input is in digital form we can use the values to control our vehicle.

Input to the analogue port can be read from BASIC using the BBC BASIC command ADVAL. The value returned by ADVAL is in the range 0 to 65520, the upper limit corresponding to an input equal to the reference voltage. Reduced input voltages will produce correspondingly smaller numbers until an input voltage of zero volts will produce a value of zero returned by ADVAL. For our simple application we are only really interested in the two limiting values. The channel read by ADVAL is determined by the number in a bracket after the keyword. Thus ADVAL (1) will read channel 1 and return a value in the range 0 to 65520.

Analogue Reflections

BBC Micro joysticks are usually supplied in pairs, going into a single connector. The analogue port pin-outs as seen from outside the machine show that joystick 2 is connected in an exactly similar way to joystick 1. Single joysticks may be connected in the place of either

ADVAL(0) performs two different functions. The least significant two bits correspond to the fire buttons on joystick 1 and joystick 2. X=ADVAL(0) AND 3 will return a value of one if joystick 1's fire button is pressed. X=ADVAL(0) DIV 256 will give the number of the channel that last completed an A-to-D conversion.

As conversion of each analogue input channel takes about 10 milliseconds, then to process each of the four channels will take 40 milliseconds. In our application we use channels 1 and 2 only. We can cut down on wasted conversion time by specifying the channels that require conversion. This can be done by using *FX16,2, which enables channels 1 and 2 but disables channels 3 and 4.

The following program combines all this information to control a twin-motor Lego car.

```
10 REM BBC JOYSTICK CONTROL
20 DDR=&FE62:DATREG=&FE60
30 ?DDR=255:REM ALL OUTPUT
40 REM ENABLE A-D CHANNELS 1&2
50 *FX16,2
60 REPEAT
70 PROCtest_joystick
80 UNTIL fire=1
90 END
100 :
110 DEF PROCtest_joystick
120 REPEAT
130 channel=ADVAL(0) DIV 256
140 UNTIL channel<>0:REM WAIT FOR CONVERT
150 IF channel=1 THEN PROCleft_right
160 IF channel=2 THEN PROCup_down
170 ENDPROC
180 :
190 DEF PROCleft_right
200 REPEAT
210 joyval=ADVAL(1)
220 IF joyval<100 THEN ?DATREG=9
230 IF joyval>64000 THEN ?DATREG=6
240 fire=ADVAL(0) AND 3
250 PRINT?DATREG,channel,joyval
260 UNTIL(joyval>100 AND joyval<64000) OR fire=
270 ?DATREG=0
280 ENDPROC
290 :
300 DEF PROCup_down
310 REPEAT
320 joyval=ADVAL(2)
330 IF joyval<100 THEN ?DATREG=10
340 IF joyval>64000 THEN ?DATREG=5
350 fire=ADVAL(0) AND 3
360 PRINT?DATREG,channel,joyval
370 UNTIL(joyval>100 AND joyval<64000) OR fire=
380 ?DATREG=0
390 ENDPROC
```

Exercise Answers

1) Calibration of your vehicle can be done by timing the period taken to travel various distances, typically 10 cm, 20 cm, 50 cm, 100 cm and 150 cm. By calculating the speed over each distance and averaging, a good estimate can be made for the distance travelled in a second. This value can then be used to control the vehicle over measured distances. A similar approach could be adopted for turning, selecting a number of angles and making timings. Controlling motors by switching them on and off over measured time periods can present many difficulties, not least that the structure of the controlling program is such that time intervals must be measured as accurately as possible. Differences of a few hundredths of a second in timing can produce large discrepancies in distances travelled or angles turned through. These problems can be reduced substantially by introducing reduction gearing between the motor and the driving wheels.

2) The program listing given on page 613 will allow you to steer the vehicle through the obstacle course. Retracing the pattern in reverse is a little more tricky. We must first assign a pair of variables for each direction together with its inverse. So, for example, forward and reverse are paired together.

Commodore 64

```
17 A(1)=5:B(1)=10:A(2)=10:B(2)=5
18 A(3)=6:B(3)=9:A(4)=9:B(4)=6
```

The following routines can then be added to play back the recorded sequence in reverse.

```
92 GOSUB2000:REM REVERSE REPLAY
2000 REM REVERSE REPLAY S/R
2010 FOR I=C TO 1 STEP -1
2020 FOR J=1 TO 4
2030 IF DR(I,1)=A(J) THEN POKE DATREG,B(J):J=4
2040 NEXT J
2050 T=TI
2060 IF(TI-T)<DR(I,2) THEN 2060
2070 NEXT I
2080 STOP
2090 RETURN
```

BBC Micro

```
1020  DIM DR(100,2),A(10),B(10)
1025 A(1)=5:B(1)=10:A(2)=10:B(2)=5
1026 A(3)=6:B(3)=9:A(4)=9:B(4)=6
1115 PROCreverse_replay
2000 DEF PROCreverse_replay
2010 FOR I=C TO 1 STEP-1
2020 FOR J=1 TO 4
2030   IF DR(I,1)=A(J) THEN?DATREG=B(J):J=4
2040 NEXT J
2042 TIME=0
2045 REPEAT UNTIL TIME>=DR(I,2)
2047  ?DATREG=0
2050 NEXT I
2055 PRINT"HIT C TO CONTINUE"
2060 REPEAT A$=GET$
2070 UNTIL A$="C"
2080 ENDPROC
3000 FORI=1TOC:PRINTDR(I,1),DR(I,2)
3010 NEXT
```

3) Assuming that line 7 is forward, line 6 is reverse, line 5 is left and line 4 is right:

```
10 REM BBC EXTERNAL SWITCHES
20 DDR=&FE62:DATREG=&FE60
30 ?DDR=15:REM LINES 4-7 INPUT
40 ?DATREG=0
50 PROCtest_input
60 GOTO50
70 :
80  DEF PROCtest_input
90 IF(?DATREG AND 240)=240 THEN ?DATREG=0
100 IF(?DATREG AND 128)=0THEN ?DATREG=5
110 IF(?DATREG AND 64)=0THEN ?DATREG=10
120 IF(?DATREG AND 32)=0THEN ?DATREG=6
130 IF(?DATREG AND 16)=0THEN ?DATREG=9
140 ENDPROC
```

```
10 REM CBM 64 EXTERNAL SWITCHES
20 DDR=56579:DATREG=56577
30 POKEDDR,15:REM LINES 4-7 INPUT
40 POKEDATREG,0:REM MOTORS OFF
50 REM L7 FORWD,L6 REV,L5 LEFT, L4 RIGHT
55 REM IS ONE OF THE INPUT LINES LOW?
60 IF(PEEK(DATREG)AND240)<>240THENGOSUB1000:GOTO60
70 POKEDATREG,0:GOTO60
80 :
1000 REM SCAN INPUT LINES S/R
1005 IF(PEEK(DATREG)AND128)=0THENPOKEDATREG,5
1010 IF(PEEK(DATREG)AND64)=0THENPOKEDATREG,10
1020 IF(PEEK(DATREG)AND32)=0THENPOKEDATREG,6
1030 IF(PEEK(DATREG)AND16)=0THENPOKEDATREG,9
1040 RETURN
```

JUMP LEADS

Following our examination of indexed addressing on the 6809 processor, we now consider how indirect addressing is used, and illustrate this by describing routines to write characters to a screen display.

First of all, it should be stated that indirect addressing is not a separate addressing mode in its own right but is an additional feature that may be used in combination with most other modes; it is really a further stage in calculating the effective address (the address from which the data is actually to be fetched). The effective address is calculated in any of the ways we have described (by direct access, by indexed addressing, or by effective address calculation), but if indirection is specified then the contents of the address so calculated and the next consecutive memory location are treated as an address. It is this address that becomes the final effective address, from which data is loaded.

For example, if the following values are stored:

Address	Contents
3000	40
3001	0A
400A	F2

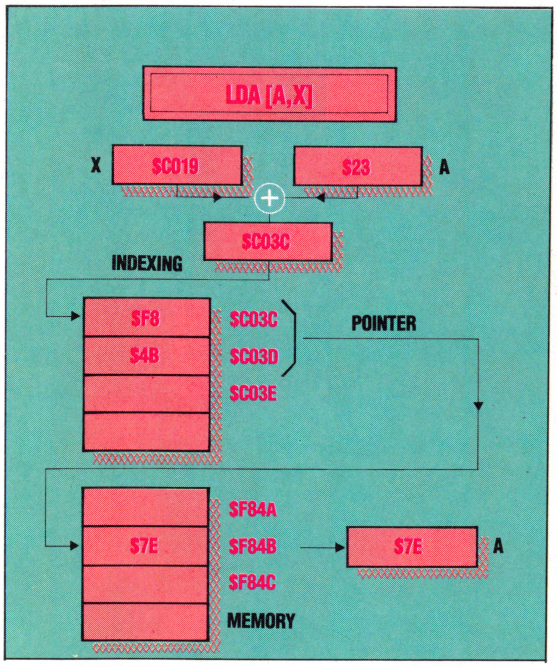
then the instruction LDA $3000 will load the value $40 into accumulator A, the effective address being $3000. Indirection is always specified by placing square brackets around the operand, so LDA [$3000] will load the value $F2 into A, the effective address being the value stored in the address that is in turn stored in $3000 and $3001 — in this case, $400A. The contents of $3000 and $3001 form a pointer or vector to the effective address, $400A. Notice the 6809 convention that addresses are stored with hi-byte before lo-byte: thus $40 is stored in $3000 and $0A is stored in $3001. This is called the hi-lo convention. The Zilog Z80 and MOS Tech 6502 processors use the opposite convention — they would require $0A (the lo-byte of the address) to be stored in $3000 and $40 (the hi-byte) to be stored in $3001.

Indirection can often be most effectively used in combination with indexed addressing. The instruction LDA [A,X], which is in indirect indexed addressing mode, will calculate an address by first adding the contents of A and X and then using the 16-bit value that is stored at this and the next location as the effective address whose contents will be loaded into A.

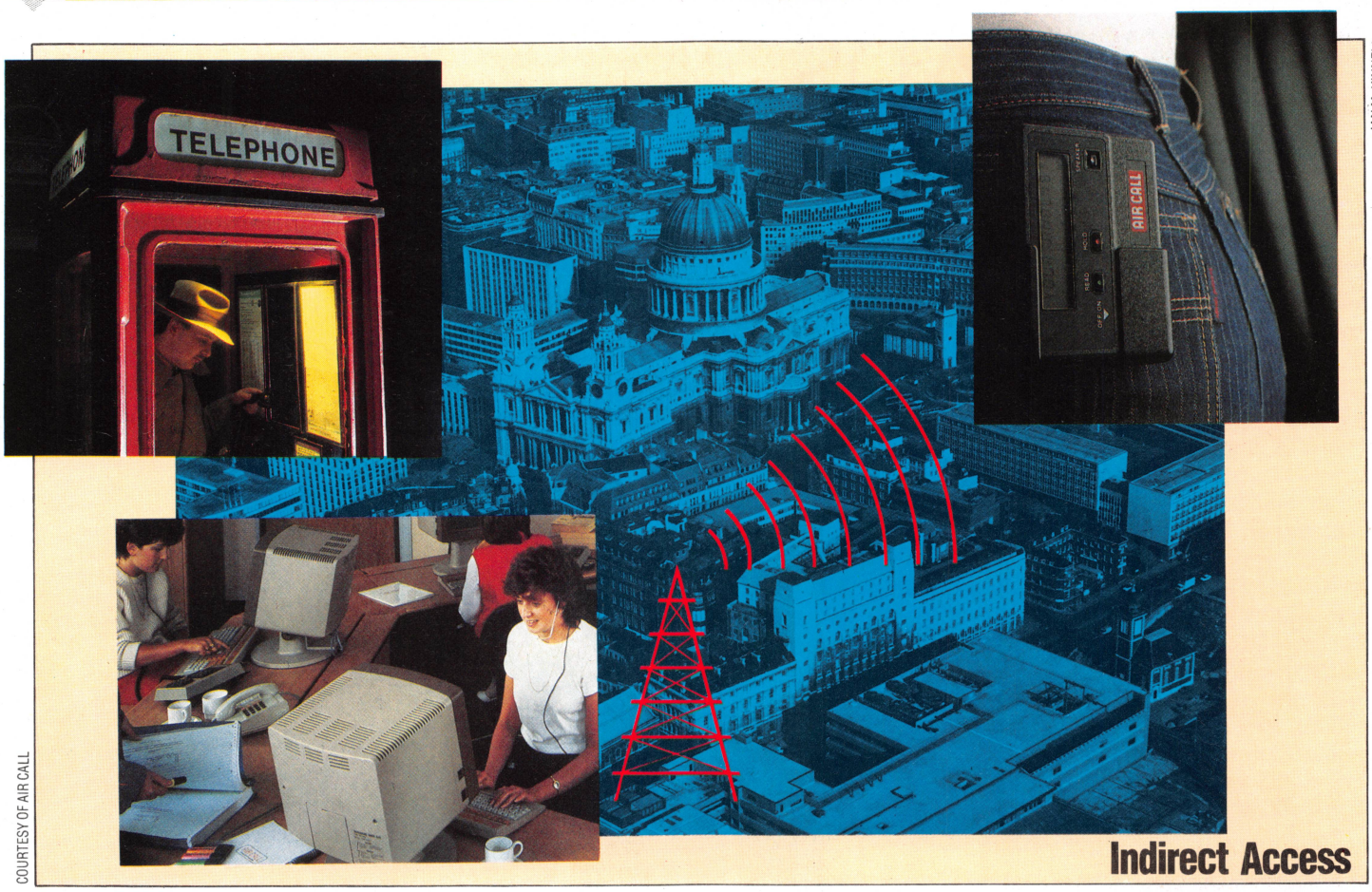
The 6809 has, in fact, less use for indirect addressing than most processors (both 6502 and Z80 programs use it frequently) because of its wealth of indexed addressing modes. There are, however, situations in which indirection can be very useful — one of these, which we will deal with at greater length in a future instalment, is the use of peripheral interface devices. Motorola processors, unlike Intel's 8080 and 8086 families, have memory-mapped I/O (Input/Output). The communications registers in the interface devices appear in the system's main memory map, and values can be stored to or loaded from them as if they were any other memory locations instead of being, effectively, a channel to the interface device. A routine to control one of these devices — for example, a print routine — needs the address of the device's interface register. If the device is relocated in the memory map, or if there is more than one device of that type, then it is much simpler to deal with this by changing one memory location that contains the address of the device communication register (a pointer to the device) rather than having to find and change every occurrence of the device address. The routine refers to the device indirectly, using the pointer.

This example illustrates the general usage of indirect addressing — when addresses that a program refers to may be changed, it is more convenient to use fixed-address pointers to refer to these locations. In this way, changes in the actual locations require only changes in the pointer contents: the program refers to the addresses indirectly.

The most common use of this technique is in a structure known as a *jump table*, which is simply a table of pointers. Any operating system contains a large number of useful routines that carry out the elemental functions of the machine — for example, reading a character from the keyboard or displaying a



Indirect Indexed Addressing
The argument [A,X] of the LDA instruction is in brackets, meaning that the contents of X ($C019 here) are to be added to the contents of A ($23), giving a 16-bit address ($C03C). This byte and the next ($C03D) are to be treated as a pointer to the effective load address ($F84B) whose contents ($7E) are finally loaded into A. Because X is added to A before the indirect access, this is known as pre-indexed indirection; the alternative, post-indexing, requires that the indirect address be calculated before the indexing takes place



IAN McKINNELL

COURTESY OF AIR CALL

Indirect Access

Vital though indirect addressing is to computer operations, it is difficult to find examples of the technique in real life. A reasonable analogy, however, is a radio paging service. When someone wants to talk to a subscriber, he doesn't call them directly, because they could be anywhere. Instead, he calls the paging service, who then page the subscriber. This is a simple, flexible system in which the paging service provides indirect accessing (or addressing) of its subscribers

character on the screen. Most machine code programs will need to use these routines at some time. In most cases, these routines will be accessed by using a jump table, which means that routines pointed to by the jump table vectors may be changed, or relocated in memory, without changes being needed in the programs that use them. In other words, such routines are always accessed indirectly, through the appropriate pointers in the jump table. When a new version of an operating system is designed, or an updated ROM produced, it is rare for these primitive OS routines to remain in their original positions; but if the jump table remains in position, with its pointers altered so that they reflect the new OS routine addresses, any software written for the old operating system that uses the jump table will run unchanged on the new system.

A common technique used in many operating systems is to have one entry point and to make all subroutine calls to this one address. One of the CPU registers is used in addition to pass a function code that is used to determine which subroutine will be called. This function code is used as an index or offset into the appropriate vector of the jump table, and control is transferred through this pointer to the desired routine.

As an example, let us suppose that we have four Kbytes of ROM, located at $F000, the first 256 bytes of which ($F000 to $F0FF) contain a table of up to 128 addresses of subroutines stored in the ROM. The entry routine (the address by which all the OS routines are addressed) is located at $F100, and this expects a function code in the range 0 to 127 to be stored in accumulator B; this code is used by the entry routine to pass control to the appropriate subroutines and thence back to the calling program once execution is complete. The calling routine for function number 1 is:

```
LDB #1        put the function code in B
JSR $F100     call the entry subroutine
```

The entry routine itself is:

```
LDX $F000     start address of the jump table
LSLB          shift B one place to the left (equivalent to multiplying
              the contents of B by two) since each entry in the table is
              two bytes long. Thus the pointer appropriate to function
              code 1 is stored at $F002 and $F003, while the pointer
              for code 2 is at $F004 and $F005, and so on
BRA [B,X]     transfer control to the address found at the Bth position
              in the table
```

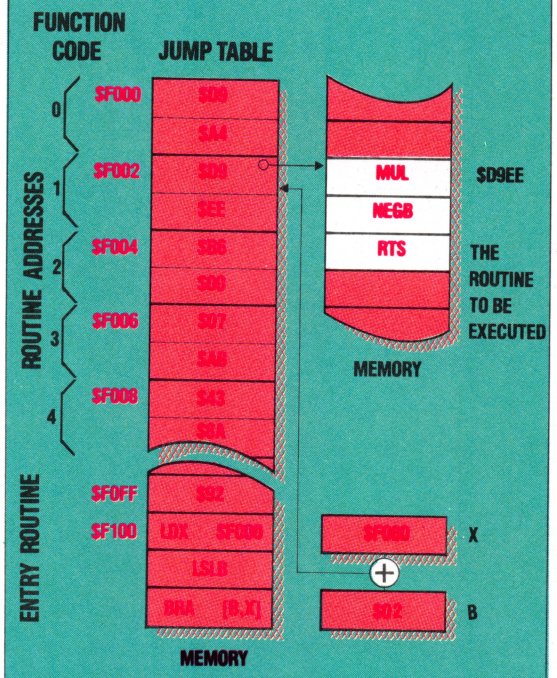
Note that the transfer to the routine is handled by a BRA (or JMP) rather than a BSR (or JSR); this is so that the RTS at the end of the OS routine will return control directly to the calling program instead of back to this entry routine.

Our next example shows a further possible use of indirect addressing in dealing with a memory-mapped display screen; on many micros the screen memory occupies a block of the main memory and may be accessed directly if extra speed is required. For simplicity, let us assume that the screen occupies a block of memory from $E000 to $E3FF, representing 16 lines of 64 characters. The position of the cursor is a 16-bit value in this range, and is located at $E400. The first subroutine clears the screen by writing a space character (ASCII code 32) at each character position. The second subroutine will write the

character passed in A to the screen at the current cursor position, unless that character is a carriage return (ASCII 13), in which case it will clear the rest of the line and position the cursor at the beginning of the next line. The cursor is represented by the underline character ('_') in this example.

```
SPACE     EQU 32          ASCII code for space
CR        EQU 13          ASCII code for carriage return
HOME      EQU $E000       Start of screen memory
LENGTH    EQU 1024        Size of screen memory (16 lines × 64
                          characters = 1024)
CURSOR    EQU $E400       $E400 and $E401 together point to
                          the current address of the cursor in
                          screen memory area
          ORG $1000
CURCHR    FCB 95          Underline character (ASCII 95)
***Subroutine to clear screen***
          LDA #SPACE      Space character in A
          LDX #HOME       Point cursor to beginning of screen
          STX CURSOR      Store current cursor position at
                          CURSOR (i.e. $E400, $E401)
          LDB #LENGTH     Size of screen in B
LOOP1     STA [CURSOR]    Store a space in current cursor
                          position
          INC CURSOR      Increment cursor position
          DECB            Decrement amount of screen
                          memory remaining between cursor
                          position and end of screen memory
          BGT LOOP1       Next space, until no more screen
                          memory remains
          STX CURSOR      Cursor back to home position
          LDA CURSOR      ASCII code of cursor character in A
          STA [CURSOR]    Store cursor character in current
                          cursor position
          RTS
***Subroutine to display character in A, if displayable***
          CMPA SPACE      Space is the first printable character
                          in ASCII
          BLT NOTP        If accumulator contains ASCII value
                          less than 32, this is non-printable, so
                          GOTO NOTP
          STA [CURSOR]    Store in current cursor position
          INC CURSOR      Increment cursor position
CHKEOS    LDX #HOME       Check for end of screen
          LEAY #LENGTH,X  End of screen in Y
          CMPY CURSOR     If cursor position exceeds end of
                          screen then...
          BGT FINISH      we have reached the end of the
                          screen, so GOTO FINISH
***Subroutine to scroll screen***
SCROLL    LEAY 64,X       Y is one line length from X (end of
                          screen memory)
          LDB #LENGTH     Calculate amount to scroll
          SUBB #64        Subtract 64 from length
LOOP2     LDA ,Y+         Move characters back one line (notice
                          auto-increment — see page 618)
          STA ,X+
          DECB
          BGT LOOP2       Loop until scrolling complete
          LDD CURSOR      Cursor to start of last line
          SUBD #64
          STD CURSOR
          BRA FINISH
***Subroutine to check for carriage return***
NOTP      CMPA #CR        Is this non-printable character a
                          carriage return?
          BNE FINISH      Ignore if not
          LDD CURSOR      You can work out how this gives the
          ANDB #%11100000 start of the next line (notice binary
                          AND-mask)
          ADDD #64
          STD CURSOR
          BRA CHKEOS      Check if end of screen
FINISH    LDA CURSOR      Cursor character in A
          STA [CURSOR]    Store in current cursor position
          RTS
```

The Jump Table



The jump table in this example is a list of 128 two-byte address pointers located between $F000 and $F0FF. Each of these pointers contains the start address of a routine somewhere in memory. To execute any of these routines we need only load the B accumulator with a function code ($01, for instance) which identifies the desired routine (located at $D9EE in this example) and then JSR to the so-called 'entry routine', start address $F0FF here. We assume that these routines are in ROM (because they are part of some ROM-based software such as the operating system) so we will be able to look up the function code and the entry routine start address in the programmer's manual.

The entry routine multiplies the function code by two, and uses it as an offset to the table start address to find the desired routine's address pointer: the pointer to routine $01 is located at $F002, for example (=$F000+2×$01), routine $02's pointer is at $F004 (=$F000+2×$02), and so on. The pointer is then used by the entry routine in an indirect branch instruction to pass control to the actual routine at $D9EE. Notice that the entry routine branches to (rather than calls) the execution routine, so that when RTS is encountered, control will pass back to the point in the program from which the entry routine was first called.

The advantage of a jump table (especially when used with an entry routine) is that it allows programmers to redesign and relocate the routines that it addresses, while still permitting programs written before such revisions to run on the new system: the function codes and the address of the entry routine are kept constant throughout the life of the system, but the contents of the jump table address pointers (and even the location of the jump table itself) may change at will.

ORIGIN OF THE SPECIES

Originally produced for the arcades, variations of Pacman have appeared on many home computers — although Atarisoft has guarded its copyright jealously and these have all differed, at least cosmetically, from the original. Atarisoft has now produced 'official' versions for the Spectrum and Vic-20.

Pacman was the original arcade maze game, and set the formula that has since been followed by home computer software like Atic Atac and Jet Set Willy. In such games the central character must travel through a network of corridors or rooms, all the while picking up treasure and avoiding the various hazards along the way. Under certain conditions, it is possible for the player to turn the tables on the attacking monsters — in Sabre Wulf, for example, running over certain orchids makes the player invisible to the creatures in the jungle. This idea was borrowed from adventure games, in which the possession of a magic sword or similar item gives the player an advantage.

In Pacman, the title character is placed at the centre of a maze, and the player must attempt to guide this figure around the screen while swallowing the dots that litter the path. A careful eye must be kept on the pursuing ghosts, which exhibit a dogged determination to trap the Pacman in the various dead-ends throughout the maze. Eating a 'power pill' allows the player to chase and eat the ghosts, thus scoring extra points, and pieces of fruit make random appearances and add to the score if eaten.

The comparison of different versions of the same game may be a little unfair — after all, a game designed for a particular machine will obviously make the most of that micro's capabilities. This is particularly true for games designed specifically for Atari machines, which have an enviable reputation for high-quality, if expensive, arcade-type software. But it must be said that the new versions of Pacman do not match up to the original.

Pacman on the Atari home computers features a large maze with excellent sound and graphics, smooth sprite movement and various skill levels, and, allowing for the fact that the game is not played on a dedicated machine, this version is a faithful replica of the arcade game. The Spectrum and Commodore versions are not in the same league. With the Vic-20, the programmers were faced with the usual problem of trying to fit a quart into a pint pot — the maze is only a quarter of the size of its Atari counterpart, whereas the sprites are twice as big as they were originally, and the cramped space makes it very difficult to avoid the ghosts. However, the graphics are well-defined, the movement is smooth and the sound is as good as that on the Atari version. The Spectrum Pacman gives fuller instructions, and allows the player the option of keyboard control instead of the joysticks that are required for Atari and Commodore versions. There is little that the Atarisoft programmers could have done about the Spectrum's pathetic sound facilities, but the jerky graphics are extremely disappointing — although the maze is very similar to the original, the flickering movement makes playing the game rather like watching a silent film. The Vic version's limitations are explained by the hardware restrictions, but it is difficult to understand why the Spectrum game is so poor.

When Pacman first made an appearance in 1980, it very quickly became a sensation, but today it appears somewhat dated. Atarisoft was quick to threaten legal action against rival software companies that brought out Pacman-type games, but the company has delayed too long in releasing these 'official' versions for other machines. A few years ago, Atarisoft could have sold thousands of copies of Pacman, but today there are many better arcade-style games on the market, and it would be a pity if Spectrum and Vic owners bought the Atarisoft Pacman in the mistaken belief that they were getting a close copy of the arcade original.

Pacman: For all Atari computers and Commodore Vic-20, £9.99. For the Spectrum, £7.99.
Publishers: Atari Corporation UK Ltd., Atari House, Railway Terrace, Slough, Berkshire.
Authors: Atari
Joysticks: Required
Format: Atari and Vic-20, cartridge; Spectrum, cassette

LIZ DIXON

DATABASE

Here, courtesy of Zilog Inc., we produce another part of the Z80 programmers' reference card.

Exchange Group

		IMPLIED ADDRESSING				
		AF'	BC', DE' & HL'	HL	IX	IY
IMPLIED	AF	08				
	BC, DE & HL		D9			
	DE			EB		
REGISTER INDIRECT	(SP)			E3	DD E3	FD E3

Block Transfer Group

HL points to source
DE points to destination
BC is byte counter

DESTINATION		SOURCE		
		REG. INDIR.		
REG. INDIR	(DE)	(HL)		
		ED A0	'LDI'—Load (DE) ← (HL) Inc HL & DE, Dec BC	
		ED B0	'LDIR'—Load (DE) ← (HL) Inc HL & DE, Dec BC, Repeat until BC = 0	
		ED A8	'LDD'—Load (DE) ← (HL) Dec HL & DE, Dec BC	
		ED B8	'LDDR'—Load (DE) ← (HL) Dec HL & DE, Dec BC, Repeat until BC = 0	

Block Search Group

HL points to location in memory to be compared with accumulator contents
BC is byte counter

SEARCH LOCATION	
REG. INDIR.	
(HL)	
ED A1	'CPI' Inc HL, Dec BC
ED B1	'CPIR'—Inc HL, Dec BC repeat until BC = 0 or find match
ED A9	'CPD'—Dec HL & BC
ED B9	'CPDR'—Dec HL & BC Repeat until BC = 0 or find match

Exchange, Block Transfer, and Search Groups

Mnemonic	Symbolic Operation	S	Z		H		P/V	N	C	Opcode 76 543 210	Hex	No.of Bytes	No.of M Cycles	No.of T States	Comments
EX DE, HL	DE ↔ HL	•	•	X	•	X	•	•	•	11 101 011	EB	1	1	4	
EX AF, AF'	AF ↔ AF'	•	•	X	•	X	•	•	•	00 001 000	08	1	1	4	
EXX	BC ↔ BC' DE ↔ DE' HL ↔ HL'	•	•	X	•	X	•	•	•	11 011 001	D9	1	1	4	Register bank and auxiliary register bank exchange
EX (SP), HL	H ↔ (SP + 1) L ↔ (SP)	•	•	X	•	X	•	•	•	11 100 011	E3	1	5	19	
EX (SP), IX	IX_H ↔ (SP + 1) IX_L ↔ (SP)	•	•	X	•	X	•	•	•	11 011 101 11 100 011	DD E3	2	6	23	
EX (SP), IY	IY_H ↔ (SP + 1) IY_L ↔ (SP)	•	•	X	•	X	•	•	•	11 111 101 11 100 011	FD E3	2	6	23	
LDI	(DE) ← (HL) DE ← DE + 1 HL ← HL + 1 BC ← BC − 1	•	•	X	0	X	↕ ①	0	•	11 101 101 10 100 000	ED A0	2	4	16	Load (HL) into (DE), increment the pointers and decrement the byte counter (BC)
LDIR	(DE) ← (HL) DE ← DE + 1 HL ← HL + 1 BC ← BC − 1 Repeat until BC = 0	•	•	X	0	X	0	0	•	11 101 101 10 110 000	ED B0	2 2	5 4	21 16	If BC ≠ 0 If BC = 0
LDD	(DE) ← (HL) DE ← DE − 1 HL ← HL − 1 BC ← BC − 1	•	•	X	0	X	↕ ①	0	•	11 101 101 10 101 000	ED A8	2	4	16	
LDDR	(DE) ← (HL) DE ← DE − 1 HL ← HL − 1 BC ← BC − 1 Repeat until BC = 0	•	•	X	0	X	0	0	•	11 101 101 10 111 000	ED B8	2 2	5 4	21 16	If BC ≠ 0 If BC = 0
CPI	A − (HL) HL ← HL + 1 BC ← BC − 1	↕	↕ ②	X	↕	X	↕ ①	1	•	11 101 101 10 100 001	ED A1	2	4	16	
CPIR	A − (HL) HL ← HL + 1 BC ← BC − 1 Repeat until A = (HL) or BC = 0	↕	↕ ②	X	↕	X	↕ ①	1	•	11 101 101 10 110 001	ED B1	2 2	5 4	21 16	If BC ≠ 0 and A ≠ (HL) If BC = 0 or A = (HL)
CPD	A − (HL) HL ← HL − 1 BC ← BC − 1	↕	↕ ②	X	↕	X	↕ ①	1	•	11 101 101 10 101 001	ED A9	2	4	16	
CPDR	A − (HL) HL ← HL − 1 BC ← BC − 1 Repeat until A = (HL) or BC = 0	↕	↕ ②	X	↕	X	↕ ①	1	•	11 101 101 10 111 001	ED B9	2 2	5 4	21 16	If BC ≠ 0 and A ≠ (HL) If BC = 0 or A = (HL)

NOTES ① P/V flag is 0 if the result of BC − 1 = 0, otherwise P/V = 1
② Z flag is 1 if A = (HL), otherwise Z = 0

Flag Notation • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown.
↕ = flag is affected according to the result of the operation